Simulação Compilada para Arquiteturas Descritas em ArchC

Marcus Bartholomeu

Tese de Doutorado

Instituto de Computação
Universidade Estadual de Campinas

Simulação Compilada para Arquiteturas Descritas em ArchC

Marcus Bartholomeu[1]

Setembro de 2005

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo (Orientador)
- Prof. Dr. Paulo César Centoducatte
  Instituto de Computação - UNICAMP
- Prof. Dr. Ricardo Pannain
  Instituto de Computação - UNICAMP
- Prof. Dr. Luiz Cláudio Villar dos Santos
  Departamento de Informática e Estatística - UFSC
- Prof. Dr. Hans-Jorg Andreas Schneebeli
  Departamento de Engenharia Elétrica - UFES
- Profa. Dra. Edna Natividade da Silva Barros (Suplente)
  Centro de Informática - UFPE
- Prof. Dr. Ricardo de Oliveira Anido (Suplente)
  Instituto de Computação - UNICAMP

[1]**Bolsista da FAPESP pelo processo 01/09424-0.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecário: Maria Júlia Milani Rodrigues – CRB8a / 2116

Bartholomeu, Marcus
B283s Simulação compilada para arquiteturas descritas em ArchC / Marcus Bartholomeu -- Campinas, [S.P. :s.n.], 2006.

Orientadores : Rodolfo Jardim de Azevedo; Guido Costa Souza de Araújo
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1 Arquitetura de computador. 2. Simulação (Computadores). 3. Multiprocessadores. I. Azevedo, Rodolfo Jardim de. II. Araújo, Guido Costa Souza de. III. Universidade Estadual de Campinas. Instituto de Computação. VI. Título.

Título em inglês: Compiled simulation for computer architectures described with ArchC

Palavras-chave em inglês (Keywords): 1. Computer architecture. 2. Computer simulation. 3. Multiprocessors.

Área de concentração: Simulação de Arquiteturas Computacionais

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Paulo Cesar Centoducatte (IC-UNICAMP)
Prof. Dr. Ricardo Pannain (IC-UNICAMP)
Prof. Dr. Luiz Cláudio Villar dos Santos (CT-UFSC)
Prof. Dr. Hans-Jorg Andreas Schneebeli (DEE-UFES)

Data da defesa: 11/11/2005

Simulação Compilada para Arquiteturas Descritas em ArchC

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Marcus Bartholomeu e aprovada pela Banca Examinadora.

Campinas, setembro de 2005.

Prof. Dr. Rodolfo Jardim de Azevedo
(Orientador)

Prof. Dr. Guido C. S. de Araújo
(Co-orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 11 de novembro de 2005, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Luiz Cláudio Villar do Santos
CT/UFSC

Prof. Dr. Hans Jorg Andreas Schneebeli
DEE/UFES

Prof. Dr. Paulo Cesar Centoducatte
IC - UNICAMP

Prof. Dr. Ricardo Pannain
IC – UNICAMP

Prof. Dr. Rodolfo Jardim de Azevedo
IC – UNICAMP

“O futuro pertence àqueles que acreditam na beleza de seus sonhos.”
Eleonor Roosevelt

Resumo

O simulador é uma das ferramentas mais importantes para o desenvolvimento de uma nova arquitetura computacional. Entre as vantagens que ele apresenta, destacam-se a flexibilidade e baixo custo. Os primeiros simuladores eram criados manualmente, uma prática muito propensa a erros. Atualmente, Linguagens de Descrição de Arquiteturas (ADLs) facilitam a geração dessas ferramentas.

O foco deste trabalho é a pesquisa em técnicas de simulação rápida utilizando a ADL ArchC. Partindo do estado da arte nesta área, a simulação compilada, conseguiu-se melhorar ainda mais o desempenho dos simuladores de conjunto de instruções. Duas otimizações foram propostas. A primeira simula atomicamente os blocos básicos, sem nenhum tipo de teste entre as instruções, oferecendo um ganho de desempenho médio de 70%. A segunda otimização calcula antecipadamente o alvo da maioria das instruções de salto, permitindo o controle do fluxo de execução pelo simulador. Esta otimização é um passo adicional à primeira, oferecendo um ganho de desempenho combinado médio de 180%.

Além da simulação rápida, também foi desenvolvida uma metodologia para que simuladores de arquiteturas possam interagir com a máquina hospedeira permitindo, por exemplo, o acesso a arquivos locais e a passagem de parâmetros de linha de commando. Também foi definida uma interface para acesso a dados externos que permitiu a implementação de um protótipo para simulação de arquiteturas com múltiplos processadores. Esta interface de dados será integrada à ArchC e permitirá a co-simulação de blocos genéricos descritos em SystemC.

Abstract

The simulator is one of the most important tools to design a new computer architecture. It has many advantages, the most important are flexibility and low cost. The first simulators were created manually, which was an error-prone practice. Nowadays, Architecture Description Languages (ADLs) simplifies the generation of these tools.

This work focus on the research of new fast simulation techniques using the ArchC ADL. Beginning from the state-of-art in this area, the compiled simulation, is was possible to speed-up the instruction set simulation performance even higher. Two optimizations were proposed. The first one simulates basic blocks atomically, without any condition test between instructions, and speed-up the simulation by 70% in average. The second optimization anticipates the majority of target address calculation for jump instructions, allowing the flow control to be done by the simulator. This second optimization is an improvement to the first one, and provides an speed-up of 180% in average.

Besides the research of fast simulation techniques, a methodology was created to allow architecture simulators to interact with the host machine, which makes it possible, for example, to access local files and take options from command-line. Also, an interface were defined to access external data which allows a propotype imlementation of a multiprocessor architecture simulator. This interface will be integrated to ArchC to achieve co-simulation capability for generic blocks described in SystemC.

Agradecimentos

Agradeço a Deus pela inspiração para fazer este trabalho. Agradeço principalmente a meus pais Luiz e Eliane e a meu irmão Ricardo, por eles sempre me incentivarem durante estes anos.

Agradeço ao meu orientador, Rodolfo Azevedo, e ao meu co-orientador, Guido Araújo, pelas trocas de idéias e pela ajuda para definir o tema de pesquisa. Agradeço também meu primeiro orientador, Ricardo Pannain, pela ajuda no meu primeiro ano em Campinas e o Sandro Rigo, por ter iniciado os trabalhos com ArchC.

Agradeço ainda as agências financiadoras que proporcionaram bolsa de estudos para eu me manter financeiramente. Obtive bolsa da CAPES no primeiro ano e da FAPESP (01/09424-0) nos anos seguintes, fechando o último semestre com uma bolsa BIPED.

São muitas as pessoas sem as quais esta Tese não seria possível. Vou tentar me lembrar da maioria delas aqui. É muito provável que eu esqueça alguém, portanto peço a meus amigos de jornada que não se sintam preteridos por não estarem aqui.

Agradeço:

a todas as gerações dos Pistolinhas (Triste, Zeh+Carmem, Baiano, Cláudio Guido, Daniel, Cleo), por estarem sempre presentes e por me concederem o título Pistolinha Honorário;

aos amigos do LSC (Thiago, Alexandro, Renon, Klein, Billo, Borin, Wesley, Guilherme, Desirée, Shiroma, Marília), pelas discussões engrandecedoras;

às gurias do IC (Amanda, Larissa, Marília, Thaisa, Ju's (do Borin e do balde), Silvana, Silvania), pela amizade;

à velha guarda (Augusto, Pará, Chenca, Tchê Felipe), pelo divertimento extra-IC;

ao pessoal de Recife (Edson, Cristiano, Pablo, Abel, Edna), pela troca de idéias;

a todos os amigos do extinto Oficina Coral Unicamp e aos do novo coral Oficina do Canto, pelos momentos felizes;

a todos os demais amigos que contribuíram para a melhoria da minha vida pessoal e acadêmica.

Grande abraço a todos,

Bartho

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A simulação é um dos métodos mais usados para avaliação de desempenho de um sistema. Entre as vantagens que ela apresenta, destacam-se a flexibilidade e baixo custo. A aplicação da idéia de simulação na área de arquiteturas de computadores é bem antiga, os primeiros trabalhos publicados sobre este assunto datam da década de 70, como a técnica de *threaded code* [1].

Os simuladores são feitos após um estudo minucioso da arquitetura que se deseja simular utilizando as estruturas de dados mais otimizadas para cada caso. Cada componente do processador deve ser programado e testado a partir do zero. Este processo pode levar meses para ser concluído, e ainda tem a desvantagem de ser muito propenso a *bugs*, uma vez que a programação de um simulador é um processo complexo e o tamanho do código é bem extenso. Depois de concluído, o simulador é executado com um bom desempenho, já que foi programado e otimizado manualmente. Porém, uma pequena mudança na especificação da arquitetura pode causar uma grande mudança no código fonte e novamente se vão semanas ou meses para atualizá-lo.

As Linguagens de Descrição de Arquiteturas (ADL) surgiram com o objetivo de facilitar a geração automática de ferramentas para uma arquitetura descrita. As ADLs são linguagens formais em que a maioria dos aspectos de uma arquitetura podem ser descri-

tos de forma facilitada e rápida. O interpretador desta linguagem deve ser capaz de criar um simulador para a arquitetura e outras ferramentas necessárias, como um montador, um compilador C, um depurador, etc. Este interpretador é certamente mais difícil de ser elaborado do que um único simulador, mas deve ser criado apenas uma vez e pode ser reaproveitado para as mais diversas arquiteturas, o que facilita muito o trabalho da equipe de desenvolvimento. Uma descrição em uma ADL pode ser feita em poucas semanas e as novas ferramentas para a arquitetura descrita são derivadas automaticamente em segundos.

A chegada das ADLs proporcionou uma melhoria adicional para o desenvolvimento de novas arquiteturas: a possibilidade de testar diversas configurações rapidamente. Esta característica é decisiva para criação de processadores mais rápidos do que os da concorrência e ainda em um tempo reduzido. Entre as configurações que podem ser testadas estão a profundidade do *pipeline*, o número de unidades funcionais, a hierarquia de memória, etc. São mudanças significativas na especificação da arquitetura, mas que podem ser facilmente descritas nas ADLs. Esta manipulação é chamada de Exploração do Espaço de Projeto (ou, em inglês, *Design Space Exploration*, DSE).

1.1 Objetivos

Uma das ferramentas mais importantes para o desenvolvimento de uma nova arquitetura é o simulador. Ele é o meio mais rápido de testar novos conceitos antes mesmo da realização de um protótipo da arquitetura.

Os primeiros simuladores gerados automaticamente com a ajuda de ADLs usaram uma técnica de simulação chamada interpretada, que imita o comportamento do hardware: primeiro busca bytes na memória, depois decodifica a sequência de bytes como uma instrução e só então a executa. Com a crescente complexidade dos sistemas, esta técnica

passou a ter um desempenho insuficiente, o que fez crescer a pesquisa nesta área para encontrar um método mais rápido de simulação.

Uma nova técnica foi idealizada para simulação de arquiteturas. Ela consiste em pré-calcular algumas operações em tempo de compilação e guardar seus resultados, evitando repetir os mesmos cálculos durante a simulação. A esta técnica se deu o nome de *simulação compilada*.

O foco deste trabalho é a pesquisa em técnicas de simulação rápida. Partindo do estado da arte nesta área, a simulação compilada, conseguiu-se melhorar ainda mais o desempenho dos simuladores de conjunto de instruções.

Neste trabalho são propostas duas otimizações. A primeira leva em conta a propriedade que blocos básicos têm de terem somente um ponto de entrada e um ponto de saída. As instruções contidas nos blocos básicos são executadas atomicamente, sem nenhum tipo de interferência ou controle pelo simulador. Esta otimização oferece um ganho de desempenho médio de 70%.

A segunda otimização calcula, antecipadamente, o alvo da maioria das instruções de salto. Nesta otimização, o controle do fluxo de execução é todo feito pelo simulador, o que permite que uma grande parte deste fluxo seja pré-calculada. Esta otimização é um passo adicional à primeira, oferecendo um ganho de desempenho combinado médio de 180%.

Além da simulação rápida, também foi desenvolvida uma metodologia para que simuladores de arquiteturas possam interagir com a máquina hospedeira. A possibilidade de comunicação de dados entre as duas máquinas permite que pelo menos duas funcionalidades importantes sejam implementadas nos simuladores: a leitura e escrita de arquivos, permitindo que a entrada e saída de dados tenham tamanho arbitrário; e o repasse de parâmetros de linha de comando.

No final do trabalho, ainda foram pesquisadas as arquiteturas de múltiplos processadores para que os simuladores gerados por ArchC dessem suporte a esta nova tendência.

Um protótipo de simulador para esta arquitetura foi desenvolvido. Este protótipo ainda não dá suporte à simulação de blocos genéricos descritos em SystemC. Esta funcionalidade está sendo desenvolvida por um aluno de mestrado partindo da interface de dados desenvolvida neste trabalho.

1.2 Organização do trabalho

O restante desta tese está organizada da seguinte forma:

O **Capítulo 2** resume diferentes abordagens relacionadas à simulação de conjunto de instruções. Comenta sobre as primeiras técnicas usadas na geração de simuladores, a facilidade de redirecionamento introduzida com o uso de Linguagens de Descrição de Arquiteturas (ADL) e técnicas recentes para aumento de desempenho dos simuladores. A última seção revisa a área de sistemas multiprocessados em um *chip* e alguns trabalhos de simulação destes sistemas.

O **Capítulo 3** propõe uma metodologia para possibilitar que simuladores de arquiteturas utilizem chamadas do sistema operacional, se comunicando com a máquina hospedeira. A técnica proposta é genérica para ser aplicada a qualquer simulador de conjunto de instruções.

O **Capítulo 4** discute os detalhes de implementação da técnica de simulação compilada em ArchC e apresenta a principal contribuição científica do presente trabalho de doutorado. Duas novas técnicas de otimização para a simulação compilada estática são propostas e implementadas em ArchC.

O **Capítulo 5** comenta que os sistemas com múltiplos processadores em um *chip* vão se tornar cada vez mais usuais tanto em computadores pessoais quanto em sistemas embarcados. Mostra ainda os trabalhos iniciais para que o ArchC dê suporte a simulação destes

sistemas. A última seção trata da integração de simuladores ArchC com blocos de lógica fechados (caixa preta) em SystemC, que será oficializada na versão 2.0 do ArchC.

O **Capítulo 6** conclui esta tese resumindo e mostrando a importância das contribuições deste trabalho. As publicações obtidas são enumeradas e alguns propostas de trabalhos futuros são sugeridas na última seção.

Capítulo 2

Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados às áreas abordadas na presente tese de doutorado. A Seção 2.1 apresenta os primeiros trabalhos para criação de simuladores de conjuntos de instruções. A Seção 2.2 comenta como as Linguagens de Descrição de Arquiteturas (ADL) facilitaram o desenvolvimento de simuladores, discute as ADLs existentes e como elas se comparam com ArchC. A Seção 2.3 volta a expor os trabalhos feitos na área de simulação de arquiteturas, agora com ênfase na técnica de simulação compilada e criação automática de simuladores redirecionáveis através de descrições em uma ADL. A última seção deste capítulo é a Seção 2.4, que contém os trabalhos na área de sistemas integrados com mais de um núcleo processador (sistema *multicore*).

2.1 Primeira geração de simuladores

Inicialmente, os estudos de arquiteturas para descoberta de defeitos de projeto eram feitos com o auxílio de analisadores lógicos ligados aos conectores do processador a ser analisado. Os sinais dos pinos externos do *chip* podiam ser visualizados com esse equipamento. Esta tecnologia apresentava como principal desvantagem a limitada observabilidade e controlabilidade na interface do circuito integrado. O estado interno do processador não podia ser visualizado e a quantidade de informações extraídas era muito grande e difícil

de ser analisada.

Uma tentativa de solução para este problema foi fazer uma análise do processador por *software*. Neste caso, uma ajuda do hardware é necessária para execução passo a passo de um programa. Após a execução de cada instrução do programa em teste, uma rotina de interrupção é chamada para verificar o estado do processador. Esta técnica de análise é chamada de execução direta. A execução direta não pode ser chamada de simulação, pois possui diversas restrições quanto as informações que podem ser extraídas do processador. Muito cuidado deve ser tomado para não executar instruções que alterem o estado do processador como visto pelo programa em teste.

A primeira técnica que cria uma camada de abstração de um novo processador sobre um processador físico, fazendo realmente uma simulação, é chamada de *threaded code*.

2.1.1 *Threaded code*

A técnica de *threaded code* foi apresentada por J.R. Bell em 1973 [1] como uma alternativa de implementação de interpretadores para máquinas virtuais. Cada instrução da máquina virtual é mapeada em uma subrotina que executa a operação desejada. Um programa a ser simulado é um vetor de ponteiros para as subrotinas que representam cada instrução do programa. A execução segue o seguinte algoritmo:

1. O valor V do contador de programa (PC) é determinado.
2. A rotina iniciada na posição V da memória é executada.
3. O valor do PC é incrementado.
4. Volte ao passo 1.

Esta máquina virtual pode ser implementada eficientemente na maioria dos processadores usando um salto indireto para o endereço que contém a próxima subrotina. Este

modelo foi depois chamado de *threaded code direto* quando novas idéias de *threaded code* surgiram.

O *threaded code* indireto apareceu para que parâmetros pudessem ser codificados junto com o programa, ao custo de mais uma indireção no algoritmo acima. Um programa passa a ser codificado conforme a Figura 2.1. Note nessa figura que os dois parâmetros necessários para a subrotina que implementa a INSTR1 são codificados após o endereço dessa subrotina.

Figura 2.1: *Threaded code* indireto

As alternativas de implementação de *threaded code* de forma portável, em uma linguagem de alto nível, são as que oferecem um salto indireto:

- **Labels:** Extensão do GNU C Compiler que permite guardar em uma variável o valor do endereço de um *label*. Posteriormente pode-se saltar para o endereço contido na variável;
- **Otimização de chamadas encadeadas:** Otimização de chamadas no fim dos procedimentos que se transformam em um salto;
- **Switch:** Estrutura *switch* do C pode funcionar como uma tabela que converte o código de uma instrução da máquina virtual para o endereço de implementação da operação.

- **Sub-Rotinas:** Chamadas consecutivas de sub-rotinas.

Entre as alternativas acima, as mais rápidas costumam ser por **Labels** e por **Sub-Rotinas**, mas isso varia muito entre compiladores e entre arquiteturas. Um *benchmark* está disponível na internet[1].

Os trabalhos mais citados que utilizam a técnica *threaded code* e são da primeira geração de simuladores são o SPIM, o `g88` e o Simics, comentados a seguir. Os trabalhos recentes com simuladores, entre eles o do Projeto ArchC, também costumam usar uma variação de *threaded code* por esta técnica permitir grande portabilidade.

SPIM

SPIM [2] é um simulador dos processadores R2000 e R3000 [3] que inclui alguma emulação de Sistema Operacional e facilidades para depuração. É desenvolvido para portabilidade da máquina hospedeira (escrito inteiramente em C) e não foi otimizado extensivamente. A aplicação alvo é inteiramente decodificada para uma representação intermediária quando esta é carregada. A execução das instruções decodificadas é cerca de 25 vezes mais lenta do que o hardware.

g88

O `g88` [4] é um simulador para o processador Motorola 88000 que suporta instruções privilegiadas e tradução de endereços. Usa a técnica de *threaded interpreter* [1] para interpretar o código de máquina do 88000, que é traduzido dinamicamente, uma instrução por vez, para um código intermediário que é então armazenado em uma *cache*. Este simulador executa código mutante[2] compatível com a ABI[3] do SPARC [5].

[1] Em `http://www.complang.tuwien.ac.at/forth/threading/`

[2] *self-modifying code*

[3] *Application Binary Interface* indica como é feita a interface hardware/software de um dado processador.

O g88 pode ser executado em processadores 68000, 88000 e SPARC. Cada instrução simulada consome uma média de 20 instruções para ser executada nestes processadores.

O simulador usa os tipos de dados e representações binárias disponíveis na arquitetura hospedeira ao invés de simular estes do 88000. Programas que se baseiam em representação binária, por exemplo, do ponto flutuante, não serão executados corretamente quando o simulador rodar em uma máquina que difere no uso do ponto flutuante.

Simics

O Simics [6] é uma evolução do g88. Foi desenvolvido no Swedish Institute of Computer Science (SICS) e simula vários processadores de arquiteturas comerciais como SPARC V8, Pentium IV e Itanium, no nível funcional. Simula sistemas operacionais comerciais e programas com um nível de detalhe escolhido pelo usuário. Suporta *multi-thread* e endereçamento virtual. O desempenho depende muito do nível de detalhe requerido mas fica normalmente em torno de 20 a 200 vezes mais lento do que o hardware original.

Atualmente é um sistema proprietário da empresa Virtutech. As características atuais desta plataforma podem ser conferidas na referência [7].

2.1.2 Tradução de código binário

A tradução de um binário executável foi uma das primeiras técnicas para uma real simulação de arquitetura. Ela consiste em replicar todos os registradores que compõem o estado interno da máquina em memória. Deste modo, o simulador tem total controle sobre todo o estado do processador. Um programa é instrumentado para que seja executado em uma máquina virtual acima da máquina real. A Figura 2.2 exemplifica como um trecho de três instruções do código binário original é instrumentado, resultando em nove instruções.

Outra possibilidade é a simulação de arquiteturas com conjunto de instruções diferentes

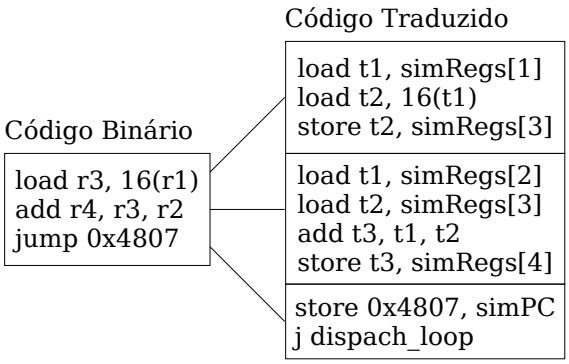


Figura 2.2: Tradução de código binário

do que existe na máquina real. Neste caso, ao invés de uma simples instrumentação, o código executável deve ser traduzido para o novo conjunto de instruções. Existem duas formas de se implementar esta idéia: estática ou dinamicamente.

A tradução dinâmica tem o *overhead* de tradução em tempo de execução, porém, este tempo é amortizado por diversas execuções da mesma instrução traduzida gravada em *cache*. Uma característica interessante dos simuladores Shade, Talisman2 (descritos a seguir) e outros que geram a tradução de código em tempo de execução é a capacidade de variação do nível de detalhe da simulação. O tipo de informação que é verificada e coletada durante a simulação e a granularidade de troca entre processadores simulados podem ser variados pela liberação de todas as instruções já traduzidas em *cache* e a consequente retradução com os novos parâmetros. Isso permite um rico conjunto de funcionalidades sem que o desempenho do simulador seja sacrificado quando elas não forem necessárias, permitindo a simulação de programas complexos, onde o pesquisador paga o custo de uma simulação detalhada somente quando os dados resultantes são importantes.

Alguns trabalhos representativos que utilizam esta técnica serão comentados nas subseções seguintes.

Shade

Shade [8] é uma máquina virtual e um *profiler* de programas. Ele emula um sistema alvo fazendo uma tradução dinâmica do código binário da máquina alvo para o código nativo da máquina hospedeira. Shade não é redirecionável, apenas algumas combinações de máquinas hospedeira/alvo foram produzidas dentre arquiteturas SPARC e MIPS. O código gerado pode contar, opcionalmente, com *profiling* para geração de *traces* da execução rodando na máquina virtual. O desempenho do Shade é de 2.3 (para ponto flutuante) a 6.2 (para inteiro) vezes mais lento do que uma execução direta em uma máquina SPARC. O *profiling* é controlado dinamicamente pelo usuário e é extensível, o usuário tem acesso ao estado da máquina virtual e pode coletar informações detalhadas sobre a aplicação que está sendo executada.

SimOS

SimOS [9] é um ambiente de simulação para sistemas com um ou mais processadores. O nível de detalhe da simulação é suficiente para execução de sistemas operacionais comerciais. Ele é capaz de simular de um modo mais rápido e com menos instrumentação a carga do sistema operacional e trocar para um modo mais detalhado durante a execução das seções interessantes da simulação.

Mesmo simulando múltiplos processadores, a simulação em SimOS é determinística. Quando os parâmetros da máquina são mantidos constantes, a execução de um programa atinge os mesmos resultados. É permitida a depuração passo a passo inclusive de tratadores de interrupção. Uma característica relacionada ao determinismo é a capacidade de gravar estados da simulação. Com a restauração de um estado salvo, a simulação continua diretamente de um estado do processador e da memória já conhecidos, sem a necessidade de iniciar um sistema operacional e carregar uma aplicação.

Dispositivos de Entrada/Saída: O SimOS opera com diversos dispositivos de en-

trada e saída para criar um ambiente de simulação mais realista. Entre os dispositivos suportados, estão o armazenamento em discos, o acesso a rede e console, e a compatibilidade com X-Windows.

Coleta de dados e classificação por Tcl-script: Com a simulação do hardware, tem-se acesso a uma quantidade enorme de informação que inclui a contagem de instruções, *cache miss*, referências a dispositivos e exceções. O SimOS facilita a coleta de informações com o uso de *scripts* na linguagem padrão Tcl. Os *scripts* podem ser executados em eventos como uma certa instrução do programa, uma certa posição de memória ou um determinado ciclo de execução. Os *scripts* têm acesso a todo o estado do simulador.

Talisman2

Talisman2[4], ou T2, é um simulador de sistemas multiprocessados que provê aos desenvolvedores de sistemas operacionais um ambiente de teste antes da disponibilidade do *hardware*. T2 usa idéias de geração de código e execução presentes nos projetos Talisman [10], Shade e SimOS. Também possui boas características para depuração como execução reversa e inserção dinâmica de módulos do usuário.

Reversão na simulação: permite a criação de *logs* de execução das instruções. Isso permite voltar para um estado anterior no tempo de simulação seguindo o *log* de trás para frente. A vantagem é permitir ao usuário responder perguntas do tipo "como o sistema chegou a este estado?" e "qual era o valor desta estrutura de dados antes de ser sobrescrita?".

Inserção dinâmica de módulos: permite ao usuário escrever, compilar e inserir código no simulador durante a execução, ou seja, sem a necessidade de reiniciar ou recompilar o simulador. O código é ligado dinamicamente com o simulador e é chamado pelo simulador em resposta a eventos. O próprio módulo do usuário tem o controle de

[4]Disponível em `http://bedichek.org/robert/talisman2/t2manual.html`

quais eventos ativam as suas rotinas. Os eventos disponíveis incluem exceções, execução ou tradução de uma instrução, ou mesmo a inserção e remoção de módulos. Esta característica pode ser usada para, por exemplo, parar a simulação assim que uma estrutura de dados se tornar inconsistente. O usuário escreve um teste de consistência como um módulo de usuário e o adiciona à simulação. Este módulo é chamado a cada evento configurado e, quando a estrutura de dados se tornar inconsistente, o módulo de usuário sinaliza o simulador para parar.

2.1.3 Execução direta por desmontagem e recompilação

Execução direta por desmontagem e recompilação consiste em adicionar um passo antes da simulação propriamente dita, onde cada instrução do programa é decodificada e reescrita em uma linguagem de alto nível, como por exemplo C. Neste passo, ainda podem ser incluídos elementos para instrumentação do código, como contagem de instruções executadas, acessos a memória, etc. Finalmente, o código de alto nível deve ser compilado para arquitetura hospedeira.

A grande vantagem desta técnica é a portabilidade, já que a máquina hospedeira pode ser qualquer máquina com um compilador para a linguagem de alto nível usada. O compilador pode ainda otimizar a simulação e a instrumentação entre um conjunto de instruções simuladas. O ponto fraco é que algumas operações sofrem perda de desempenho mesmo que elas pudessem ser mapeadas diretamente, porque o compilador não consegue utilizar a melhor operação disponível na máquina hospedeira.

Uma dificuldade adicional é descobrir o que é código e o que é dado na memória [11]. Como a técnica aplica compilação estática, códigos mutantes ou ligados dinamicamente não podem ser executados. Pelo mesmo motivo, o nível de *tracing* não pode ser mudado dinamicamente. Exceções e sinais não são normalmente suportados. A simulação fica em média 6 a 8 vezes mais lenta do que a execução nativa.

2.2 Linguagens de Descrição de Arquiteturas

Linguagens de descrição de hardware (HDL) como VHDL [12], Verilog [13] e, mais recentemente, SystemC [14] são bastante usadas para modelar e simular processadores, mas com o objetivo de desenvolver hardware. O uso destes modelos para exploração de arquiteturas e geração de ferramentas de desenvolvimento de software tem algumas desvantagens. Não existe informação sobre o conjunto de instruções do processador, como, por exemplo, a sintaxe do montador. Eles cobrem um grande número de detalhes de implementação desnecessários para avaliação de desempenho, simulação de ciclos e verificação de software. Além disso, a descrição detalhada da estrutura do hardware tem um impacto significativo na velocidade da simulação [15].

Um salto de flexibilidade ocorreu com o surgimento, nos anos 90, das linguagens de descrição de arquiteturas (ADL, do inglês *Architecture Description Language*), que permitiram uma modelagem em um nível de abstração mais alto. Desde esta época diversas ADLs foram propostas, onde as mais atuais tentam corrigir ou melhorar deficiências das primeiras ADLs propostas.

Uma ADL é uma linguagem desenvolvida para especificação de um *template* de SoCs (*System-on-Chip*). Um *template* de arquitetura inclui: blocos e componentes que residem no SoC, como eles estão conectados e a funcionalidade de cada um deles. A estrutura detalhada de cada bloco (processador, memória, hardware dedicado, etc.) não é descrita na ADL porque ela só é determinada no passo seguinte do desenvolvimento da arquitetura. Esta descrição do modelo da arquitetura pode então ser usada para geração automática (ou semi-automática) de ferramentas como compiladores, simuladores, montadores, etc. Diversas ADLs foram propostas recentemente, como veremos na próxima seção, porém nenhuma delas tornou-se um padrão. ADLs são um tópico de pesquisa relativamente novo e devem ser melhor estudadas para realização eficiente da exploração do espaço de

desenvolvimento (DSE, do inglês *Design Space Exploration*) para arquiteturas de SoC. Alguns aspectos importantes de uma ADL ideal são:

Especificação concisa e natural: As especificações em ADLs são escritas pelo projetista do sistema, portanto é importante permitir uma maneira natural e fácil de especificação. Estas características permitem reduzir o tempo e facilitar a compreensão da especificação. Além disso, uma especificação concisa (de preferência não redundante) evita erros que podem ser difíceis de serem encontrados.

Generalidade na especificação: A ADL deve ser capaz de capturar uma ampla classe de processadores, desde os mais simples RISCs até os mais complicados VLIWs, superescalares e DSPs com *datapaths* irregulares. As opções de projeto que precisam ser exploradas são vastas e só podem ser devidamente testadas através da geração automática de ferramentas de software. Hierarquias de memória compostas por DRAMs, memórias *flash*, *caches*, etc. também devem ser suportadas.

Especificação formal: ADLs devem ser linguagens formais. Através deste formalismo, o projetista do SoC pode verificar formalmente ou validar muitas propriedades que a especificação deve satisfazer nos primeiros estágios de desenvolvimento do SoC.

Geração automática de ferramentas: Uma ADL ideal deve possibilitar a geração automática de ferramentas de software de qualidade que incluem, pelo menos, um compilador com capacidade de gerar código para arquiteturas paralelas em nível de instrução (VLIW e superescalares), os chamados compiladores ILP, e um simulador com precisão de ciclos. Porém, estas ferramentas requerem informações detalhadas da arquitetura, tipicamente de uma forma que não é concisa e facilmente especificável. Portanto, tornam-se necessários procedimentos que geram automaticamente esta informação a partir da especificação ADL. Um exemplo é a especificação de conflitos de recursos entre instruções para compiladores ILP. Para o compilador gerar código otimizado, é necessária informação sobre conflito de recursos entre instruções. Tabelas de Reserva (do inglês *Reservation Tables*,

RT) são usadas por muitos compiladores ILP para descrever conflitos, entretanto, a descrição manual de RTs para cada instrução é complicada. É mais fácil exigir a especificação do *pipeline* e dos recursos do *datapath* de maneira abstrata e então gerar automaticamente RTs para cada instrução [16].

Os simuladores podem ser usados para vários propósitos, como a estimação de desempenho, consumo, análise de utilização de recursos, validação funcional do hardware, depuração do software, co-simulação do hardware e software, etc. Vários tipos de simuladores devem ser gerados para dar suporte a uma rápida DSE, como simuladores funcionais e de instruções. Os simuladores detalhados também são muito importantes (simuladores de ciclo, de fase e de pinagem).

Uma ADL deve dar suporte a todos os aspectos de um projeto de SoC, incluindo os ASIC e interfaces de I/O. Porém, muitas das ADLs existentes têm seu foco na especificação de processadores e sistemas de memória. Esta seção pretende abordar algumas características das diversas ADLs existentes, as ferramentas suportadas e as informações das arquiteturas que cada uma captura.

Dentre as ADLs existentes, pode-se verificar três grandes grupos. O primeiro tem como foco da descrição a estrutura de um processador ou como é a ligação das unidades funcionais entre si para formação do *pipeline*, como ocorre o acesso a memória e a registradores, etc. O segundo grupo se preocupa com o detalhamento do que é visto pelo usuário da arquitetura — o projeto do conjunto de instruções e a funcionalidade oferecida por ele. Estes dois grupos são comumente chamados de ADLs de estrutura e ADLs de comportamento, respectivamente. Ambos os esquemas têm seu mérito para certas classes de arquiteturas e tipos de ferramentas suportadas. Na verdade, as abordagens são ortogonais e complementares. As ADLs mais modernas fazem parte de um terceiro grupo de ADLs que dão igual importância à estrutura e ao comportamento do processador, um híbrido das duas abordagens citadas, sendo chamadas de ADLs híbridas.

2.2.1 ADLs de comportamento

As ADLs de comportamento reproduzem a visão do programador do SoC pela descrição do conjunto de instruções dos componentes programáveis do SoC.

A **nML** [17] é uma ADL de comportamento proposta pela Technische Universität Berlin. O conjunto de instruções é descrito por uma gramática de atributos. São incluídas informações sobre o comportamento, sintaxe *assembly* e código de cada instrução organizado de uma maneira hierárquica e concisa. Um ponto fraco de nML é assumir um modelo de arquitetura que não permite a descrição de unidades funcionais com *pipeline* ou multiciclos. Outro é não permitir instruções de tamanho maiores do que uma palavra. Conflito de recursos entre instruções é descrito na forma de conjuntos de combinações legais entre operações. Uma boa critica de nML é feita em [18], onde é usada para gerar um simulador, um montador e um desmontador. A nML também é usada nos simuladores SIGH/SIM [19] e CHECKERS e nos geradores de código CBC [20] e CHESS [21].

A **ISDL** [22] é uma ADL de comportamento proposta no MIT que foca em processadores VLIW. A descrição do conjunto de instruções de um processador inclui a ação de cada instrução, o formato do *assembly*, o custo (que reflete o número de ciclos, tamanho, conflito de recursos) e a temporização. A estrutura do *pipeline* não é descrita explicitamente, mas restrições de paralelismo devido a conflitos são explicitamente descritas na forma de regras booleanas que precisam ser satisfeitas para que uma instrução seja válida. A ISDL foi usada para geração automática de um compilador [23], um *assembler* e um simulador com precisão de ciclos [24]. Pode ainda ser traduzida para Verilog e, portanto, gerar código sintetizável.

A **CSDL** [25] é um conjunto de linguagens para descrição de máquinas desenvolvido na Universidade de Virginia. Consiste em quatro linguagens (SLED, λ–RTL, CCL e PLUNGE), sendo que cada uma captura uma parte específica de informação da arqui-

tetura. A SLED [26] descreve as representações do *assembly* e do binário, enquanto a λ–RTL descreve o comportamento das instruções no nível de transferência entre registradores. A CCL especifica a convenção de chamadas a funções e a PLUNGE é uma notação gráfica para especificação da estrutura do *pipeline*. A literatura não deixa claro se existem simuladores com precisão de ciclos.

2.2.2 ADLs de estrutura

As ADLs de estrutura descrevem um diagrama de blocos ou uma *net-list* da arquitetura.

A **MIMOLA** [27] é mais uma HDL do que uma ADL de estrutura. Foi proposta na Universidade de Dortmund. A estrutura do processador é modelada como uma *net-list* no nível de transferência de registradores. MIMOLA não contém informação sobre o conjunto de instruções nem sobre o *assembly* explicitamente, mas é usada no compilador RECORD [28], que consegue extrair o conjunto de instruções da descrição MIMOLA. Uma das maiores vantagens de MIMOLA é que a mesma descrição pode ser usada para síntese, simulação compilada [29], geração de teste e geração de código. Porém, não permite a descrição explícita de *pipelines* e conflitos de recursos.

O **COACH** é um sistema CAD para desenvolvimento de processadores com conjunto de instruções específicos para uma aplicação, desenvolvido na Universidade de Kyushu. Usa a HDL UDL/I [30] para descrever processadores no nível de transferência de registradores. O conjunto de instruções é extraído automaticamente [31] e usado para geração de um compilador e um simulador [32]. Assume um processador RISC e não dá suporte a *pipeline* nem ILP. Como em MIMOLA, a vantagem é poder usar a mesma descrição para síntese, simulação e compilação.

2.2.3 ADLs híbridas

As ADLs híbridas combinam as características das ADLs estruturais e de comportamento para facilitar a geração automática de ferramentas.

A **MDes** [33] é uma ADL para DSE de processadores de alto desempenho. Ela captura a estrutura e comportamento de processadores em forma de seções (formato, uso de recursos, latência, operação, registradores, etc) hierarquicamente. MDes tem boas construções de pré-processamento, o que permite descrições concisas. As restrições para execução de operações são descritas com tabelas de reserva [33]. MDes é usada em uma infra-estrutura integrada de compilação e monitoramento de desempenho chamada Trimaran.

FlexWare [34] é um sistema CAD para desenvolvimento de DSPs e ASIPs. A descrição consiste em três componentes: o conjunto de instruções, os recursos e o grafo de interconexão representando o caminho dos dados. Não fica claro como os conflitos de recursos entre instruções são especificados. A especificação de hierarquia de memória não parece ser possível. FlexWare conta com o gerador de código CodeSyn e o simulador Insulin.

EXPRESSION [35], desenvolvida na Universidade da California em Irvine, é uma ADL que permite a exploração de arquiteturas RISC a VLIW. Permite também a especificação de hierarquia de memória. A estrutura é descrita como componentes e suas interconexões. O *pipeline* é descrito como as unidades funcionais de cada estágio e suas temporizações. O comportamento do conjunto de instruções é descrito de forma hierárquica. Os conflitos de recursos não são descritos explicitamente, mas as tabelas de reserva são extraídas automaticamente [16] e passadas para um compilador ILP. Um simulador em nível de ciclos é apresentado em [36]. No começo de 2005, a linguagem EXPRESSION e a ferramenta de simulação interpretada foram colocadas em domínio público. Infelizmente a ferramenta mais interessante, a de simulação compilada dinâmica, não está no pacote

público.

LISA [37, 38], desenvolvida em RWTH Aachen, suporta geração de simuladores com precisão de bits para DSPs. Sua principal característica é a especificação do *pipeline* por operações, o que permite a modelagem de técnicas complexas de *interlock* e *bypass*. Cada instrução consiste em múltiplas operações que são definidas como transferências entre registradores durante um passo de controle, que pode ser uma instrução, um ciclo, ou uma fase do ciclo, dependendo do nível de precisão desejado. Os conflitos de recursos não são escritos explicitamente, mas através de operações que ativam ações de parada, deslocamento e descarga do *pipeline*. Atualmente LISA é uma linguagem fechada. Várias ferramentas estão disponíveis comercialmente, por exemplo montadores, compiladores e depuradores.

RADL [39] é uma extensão de LISA proposta pela RockWell Semiconductor Systems que suporta um modelo de *pipeline* mais detalhado, com *delay slots*, interrupções, *zero-overhead loops*, *hazards* e múltiplos *pipelines*. Pode gerar simuladores com precisão de ciclo (ou fase) para DSPs.

2.2.4 ArchC

ArchC [40] é a linguagem de descrição de arquiteturas desenvolvida no IC-UNICAMP que permite explorar uma diversa gama de arquiteturas. A descrição consiste em dois grandes blocos. O primeiro apresenta a estrutura do processador, a hierarquia de memória e o *pipeline*. O comportamento do conjunto de instruções é especificado no outro bloco de uma maneira hierárquica, em três níveis: comportamento geral de todas as instruções, comportamento do tipo de instrução e comportamento específico de cada instrução. O diferencial desta linguagem em relação às outras é a grande facilidade de realização de um primeiro modelo funcional. Com pouca informação sobre a arquitetura já é possível gerar um simulador do conjunto de instruções. Outra característica única é a verificação

de modelos em níveis de abstração diferentes, onde um modelo ArchC é comparado com um modelo de referência descrito em ArchC ou em SystemC.

O pré-processador da linguagem transforma a descrição ArchC em um modelo SystemC, que pode ser simulado pela técnica interpretada com precisão de instruções ou de ciclos, dependendo do nível em que processador foi modelado. Um simulador compilado também pode ser gerado automaticamente e é uma das contribuições deste trabalho de doutorado. Por enquanto, a simulação compilada ocorre somente no nível do conjunto de instruções, mas a precisão de ciclos poderá ser desenvolvida futuramente para este simulador.

Três arquiteturas RISC já foram desenvolvidas em ArchC: SPARC V8 [41], MIPS I [3] e PowerPC [42]. Outras classes de processadores como DSPs e VLIWs estão sendo estudadas para que suas particularidades sejam incluídas na linguagem.

Outra ferramenta que foi recentemente finalizada para os modelos ArchC é a de geração automática de montadores. Para utilização desta ferramenta, o modelo deve incorporar informações adicionais sobre suas instruções, como a sintaxe detalhada do *assembly* da arquitetura e a denominação dos registradores existentes. O passo seguinte nesta linha é a ferramenta de geração de compiladores, que será desenvolvida no futuro.

2.2.5 Comparação entre as ADLs

A Tabela 2.1 resume a comparação das principais características presentes em ArchC com outras linguagens de descrição de arquiteturas apresentadas nesta seção. Trabalhos em andamento são marcados com a letra **F**. LISA e EXPRESSION são os dois projetos com maior similaridade com ArchC. Uma característica interessante que é implementada somente em ArchC é a co-verificação de dois modelos em níveis de abstração diferentes, por exemplo um modelo funcional com um *cycle-accurate*. O suporte a *multi-cores* é recente em LISA e está marcado como **F** em ArchC pois o trabalho descrito no Capítulo 5 ainda

Característica	LISA	EXPRESSION	nML	ISDL	ArchC
Conjunto de Instruções	•	•	•	•	•
Precisão de Ciclos	•	•			•
Suporte a Multiciclo	•	•			•
Suporte a *pipeline*	•	•			•
Hierarquia de Memória	•	•			•
Emulação de S.O.	•				•
Hierarquia de Comportamentos					•
Co-verificação					•
Geração de Simuladores SystemC alto-nível	•				•
Geração de Montadores	•	•		•	•
Simulação Compilada	•	•			•
Redirecionamento de Compiladores	•	•		•	**F**
Suporte a *multi-cores*	•				**F**

Tabela 2.1: Comparação entre ADLs

não foi incorporado na versão oficial.

2.3 Simuladores Modernos

Tradicionalmente, a simulação de processadores era feita de uma maneira similar ao que ocorre no hardware. Os passos de busca da instrução, sua decodificação e execução são feitos sequencialmente para cada instrução. Este método é referenciado como simulação interpretada.

O significativo aumento de complexidade das novas arquiteturas configuráveis provoca um impacto negativo no desempenho dos simuladores. Levando em conta a Lei de Moore, as pesquisas em simuladores de alto desempenho foram iniciadas em 1991 [43], inicialmente em simuladores de hardware em nível de transferência entre registradores (RTL). Ela foi adaptada com sucesso a simuladores de conjunto de instruções, porém, não obteve êxito em produtos comerciais. Isso se deve ao fato de que o ganho de desempenho é obtido com o alto custo da perda de flexibilidade necessária às aplicações e arquiteturas da época. Atualmente, novas técnicas tornam os simuladores rápidos tão flexíveis quanto os

tradicionais, conforme será visto a seguir.

Diversas técnicas foram propostas para agilizar a simulação, otimizando alguns pontos no esquema da simulação tradicional. Os primeiros trabalhos trataram de amenizar o custo de decodificação ao inserir no modelo uma *cache* de instruções decodificadas. Esta otimização provoca um grande ganho de desempenho, já que em laços as instruções são decodificadas apenas na primeira rodada (se a *cache* for do tamanho do laço ou maior).

Um outro avanço foi constatar que muitas operações do simulador poderiam ser pré-calculadas antes do início da simulação. Assim, cada vez que o simulador precisasse de um resultado, ele já estaria calculado. Este raciocínio é a base para o que se chamou de simulação compilada, que é uma simulação que sofre um pré-processamento para otimização.

2.3.1 Simulação compilada

A motivação da simulação compilada é criar um simulador especializado para um dado programa. Quando se sabe de antemão qual aplicação será alvo da simulação, algumas otimizações são possíveis, por isso a simulação compilada é até duas ordens de grandeza mais rápida do que a tradicional, conforme [44] e também por resultados experimentais do Projeto ArchC.

Uma das otimizações é a realização da fase de busca e decodificação de instruções antecipadamente. Todo o programa é decodificado na fase de pré-processamento. Assim, torna-se desnecessário um decodificador em tempo de execução, o que reduz bastante o tempo de simulação. Em oposição, na simulação tradicional, cada vez que uma instrução é executada ela deve ser decodificada — note que, dentro de um laço, a mesma instrução é decodificada diversas vezes.

Outra otimização vem do fato da sequência de instruções já ser conhecida e os argumentos já estarem disponíveis antes da simulação. Técnicas de compilação estática [45]

movem o agendamento de instruções para o tempo de compilação, possibilitando efetivar *inline* das operações em que os argumentos já são conhecidos, economizando assim algumas chamadas à função no código do simulador. Esta técnica, além de reduzir a complexidade das expressões, permite que a compilação do simulador aproveite melhor outras otimizações para expressões, como eliminação e/ou propagação de constantes.

Um trabalho com simulação rápida e redirecionável (*retargetable*) foi feito usando a linguagem de descrição de arquiteturas FACILE [46]. O simulador gerado pela descrição FACILE utiliza a técnica de *fast forwarding* para atingir um desempenho relativamente alto. Essa técnica é similar a simulação compilada e usa uma *cache* com o resultado de ações do processador, indexada por um código de configuração do processador. Ações guardadas previamente podem ser reexecutadas diretamente no caso de uma configuração se repetir. Assume-se que o código do programa é estático, portanto código dinâmico não pode ser simulado. Não há resultados publicados da aplicação de FACILE em VLIW ou arquiteturas DSP irregulares.

Outra técnica de simulação rápida e *retargetable* é apresentada em [47]. Ela melhora a simulação compilada estática tradicional através do uso otimizado de recursos da máquina hospedeira. Esta utilização é conseguida pela definição de uma interface de código de baixo nível especializada para simulação de conjunto de instruções. Na prática utiliza uma linguagem própria de baixo nível, similar a um *assembly* de máquinas RISC ao invés de usar C como a linguagem intermediária de geração de código portável.

Simuladores rápidos redirecionáveis baseados em Linguagens de Descrição de Arquiteturas (ADL) foram propostos pelos projetos Sim-nML, ISDL, MIMOLA, LISA, e EXPRESSION, conforme descrito na Seção 2.2.

Nenhum dos trabalhos comentados acima, a não ser o de LISA [45], que é proprietário, faz uma simulação com precisão de ciclos usando a técnica compilada. Embora o desempenho na simulação apresentado seja muitas vezes maior do que simuladores comerciais,

que usam a técnica interpretada, muito trabalho ainda deve ser feito para melhorar este desempenho, dada a crescente complexidade dos processadores atuais e o curto espaço de tempo de desenvolvimento.

2.3.2 Limitações da simulação compilada estática

Assumir um programa estático em tempo de execução pode ser uma restrição muito grande para certas aplicações. A simulação compilada estática requer que o conteúdo da memória de programa não seja mudado durante a simulação. Isso é verdade para a maioria dos DSPs atuais que executam pequenas aplicações, já os microprocessadores normalmente executam sistemas operacionais (SO). A característica principal de todo SO é o código dinâmico em tempo de execução, o que conflita com as restrições de simulação compilada. Mesmo para DSPs, os sistemas operacionais de tempo real estão ganhando cada vez mais importância.

Existem características nas novas arquiteturas, principalmente no domínio de baixo consumo de energia, como múltiplos conjuntos de instruções para reduzir o consumo de memória e de energia. Estas arquiteturas podem passar a usar um conjunto de instruções reduzido em tempo de execução. Um exemplo é a família de processadores ARM que provê o conjunto de instruções *Thumb*. Estas trocas dinâmicas de conjunto de instruções não podem ser consideradas para um simulador compilado estático, uma vez que a seleção depende de valores só disponíveis em tempo de execução e não são previsíveis.

A simulação compilada estática é uma variação de simulação compilada com muitos trabalhos publicados atualmente. Ela apresenta um grande ganho de desempenho mas, pelos argumentos acima, pode ter sua área de aplicação restrita. Isso implica que a técnica de simulação interpretada ainda domina pelo menos para as áreas mostradas. Contudo, dada a complexidade crescente das aplicações, arquiteturas e sistemas, surge a necessidade de uma técnica de simulação rápida e flexível. Para resolver este problema

e aumentar a flexibilidade, os novos trabalhos em simulação compilada permitem que programas simulados possam sofrer modificações. Assim surgiu a simulação compilada *dinâmica*.

2.3.3 Proposta JIT-CCS

O primeiro trabalho que acaba com a limitação de código estático nos simuladores compilados veio do projeto LISA, da Universidade Aachen, na Alemanha. A proposta é levar o compilador de simulação para o tempo de execução, processando cada instrução no momento em que ela for executada. A informação extraída da instrução é guardada em uma *cache* de simulação para reuso no caso de uma próxima execução do mesmo endereço de memória. O simulador reconhece quando uma instrução já processada mudou na memória e inicia um re-processamento. Esta técnica foi denominada *just-in-time cache compiled simulation* (JIT-CCS) [48].

O processamento da instrução em tempo de execução requer uma mudança de paradigma para o compilador de simulação. Pelo menos dois aspectos devem ser analisados. Na simulação compilada tradicional, o tempo de decodificação de uma instrução não é crítico, já que não influencia o tempo de execução. Além disso, é permitido que o compilador de simulação gere código em C para uma compilação subsequente. Estes dois pontos não se aplicam a um compilador de simulação integrado. A solução do JIT-CSS é pré-compilar as funções em C que implementam as operações das instruções e, na hora da execução, selecionar as operações apropriadas. Referências às funções C selecionadas são armazenadas na *cache* de simulação e são utilizadas para execução da instrução.

O desempenho obtido com a técnica JIT-CSS melhora com o aumento do tamanho da *cache*. Cerca de 95% do desempenho do simulador compilado tradicional é obtido com uma *cache* de simulação de 4096 entradas. A memória alocada na hospedeira para simulação desta *cache* é de 2 MB, enquanto que, na simulação tradicional, a memória usada, que

depende do tamanho do programa, chega a pelo menos uma ordem de magnitude acima. Esta técnica permite um balanço entre o desempenho requerido e o uso de memória. Em um Athlon 1.2GHz com 768 MB de memória principal, a simulação executa uma média de 7 MIPS.

2.3.4 Proposta IC-CS

A técnica proposta pelo projeto EXPRESSION, da Universidade da California em Irvine, EUA, pode ser considerada como uma melhoria da JIT-CCS. São duas as principais diferenças. Primeiro, a decodificação volta a ser efetuada em tempo de compilação, mas com a possibilidade de re-decodificação no caso de uma instrução ser modificada em tempo de execução. Outra diferença é o uso de uma técnica batizada de *abstração de instrução* para gerar instruções decodificadas com otimização agressiva de modo a melhorar ainda mais o desempenho de simulação.

A *abstração de instrução* explora o fato de certas classes de instruções possuírem um valor constante para um dado campo da instrução. Como um exemplo, a maioria das instruções ARM são executadas incondicionalmente (campo de condição com valor *verdadeiro*). É um desperdício de tempo checar a condição destas instruções sempre que elas forem executadas. Para certas classes de instruções, a técnica de *avaliação parcial* pode ser usada nos casos em que se sabe de antemão valores para certos campos. O objetivo é especializar uma função com parte da sua entrada para obter uma versão mais rápida do mesmo programa. O ideal é ter funções especializadas separadas para cada tipo de instrução, mas isso pode ser inviável para algumas arquiteturas. A proposta é classificar formatos similares.

Templates da linguagem C++ são usados para implementar a avaliação parcial para cada classe de instruções. Os parâmetros do *template* são os campos da instrução. O artigo [49] mostra os algoritmos usados para decodificação, determinação do *template* e

especialização de uma instrução. A decodificação e especialização é feita em tempo de compilação e carregada em uma estrutura separada da memória principal, com ponteiros para as funções especializadas. Durante a execução, a instrução desta estrutura é comparada com a da memória principal. Caso seja igual, a função otimizada é executada, caso contrário, uma nova decodificação é feita, mas uma função genérica para a classe da instrução é usada e não uma função otimizada (como ocorre em tempo de compilação). O artigo assume que não há perda considerável de desempenho pelo fato de o número instruções modificadas em tempo de execução ser muito pequeno. Mas esta suposição depende da área de aplicação, como discutido na Seção 2.3.2. Um Sistema Operacional, por exemplo, pode trocar um programa inteiro por outro.

O desempenho obtido com a técnica IC-CS fica em torno de 12 MIPS para um processador Pentium 3 de 1 GHz e 512 MB de RAM. Para uma aplicação de 50 mil instruções, a compilação da simulação leva em torno de 15 minutos para gerar o programa decodificado e compilar este. É um tempo considerável, já que a simulação em si pode levar um tempo similar. Mas um artigo subsequente do grupo apresenta uma nova maneira de compilação para reduzir drasticamente este tempo.

Redução do tempo de compilação

A simulação compilada estática provê o melhor desempenho de simulação de conjunto de instruções. O código binário da arquitetura alvo é analisado e transformado em um código fonte que tem uma funcionalidade equivalente ao programa de entrada. Este código é otimizado e compilado em um binário da hospedeira e então executado. O problema é que quando o binário de entrada é muito grande, o tamanho do código fonte gerado aumenta proporcionalmente, o que torna a compilação do simulador muito demorada. Como toda instrução é decodificada independentemente se será ou não executada, outro problema é a grande quantidade de informação sobre todas as instruções, o que consome

muita memória. O trabalho [43] investigou meios de reduzir o tempo de compilação e chegou a particionar o código fonte em blocos menores. Esta técnica também é usada no ArchC Compiled Simulator, porém com a separação do código em funções.

Outra abordagem [50] surgiu da observação de que o número de tipos de instruções é muito menor que o números de instâncias destas. Ao invés de gerar código idêntico para instâncias de instruções, gera-se somente uma vez um código otimizado para cada tipo de instrução que existe em um programa. Em tempo de execução, existe um decodificador que seleciona dinamicamente o tipo de instrução para execução e guarda o resultado em uma *cache*. O desempenho é dito similar ao do IC-CS, embora exista o custo de decodificação em tempo de execução. A maior contribuição, no entanto, é a diminuição média no tempo de compilação de 99%.

2.3.5 Simulação com precisão de ciclos

A maioria das ADLs permitem a descrição de processadores em vários níveis de abstração. Isso reflete o processo de criação de um processador. Primeiro, uma versão de alto nível, que descreve os comportamentos das instruções, é feito. As ferramentas de compilação e simulação são geradas e inicia-se um ciclo de exploração do conjunto de instruções. O próximo passo é descrever um nível de detalhe maior da arquitetura a ponto de poder prever o seu desempenho com a precisão de ciclos executados.

Um simulador estrutural, com precisão de ciclos, permite a exploração da arquitetura (DSE) e traz benefícios como:

Comparação de arquiteturas: o simulador estrutural pode prover números comparativos para o desempenho de arquiteturas a assim auxiliar o desenvolvedor na identificação das arquiteturas mais apropriadas para o SoC.

Avaliação detalhada: os resultados da simulação de ciclos pode prover informação detalhada para avaliação de arquiteturas em termos de desempenho, energia consumida

e outras características, o que permite um ajuste fino da arquitetura.

A linguagem Sim-nML permite a modelagem de arquiteturas com *pipelines* com precisão de ciclos [51], porém não permite a descrição detalhada de processadores com mecanismos complexos de controle, como o TMS320C5000 da Texas Instruments. Esta restrição também se aplicam a ISDL [24]. EXPRESSION permite a modelagem de processadores com *pipeline* com precisão de ciclos e de bits [36]. A linguagem LISA permite *retargeting* de simuladores com precisão de ciclos para microcontroladores e DSPs [52, 53].

2.4 Sistemas Embarcados em um *Chip*

Sistemas embarcados representam a maioria do mercado de microprocessadores. Mais de 1 bilhão de microprocessadores são fabricados para uso em sistemas embarcados a cada ano [54] e este número continua a crescer rapidamente. Isto ocorre pela competição existente entre as empresas topo de linha que desenvolvem seus projetos em um intervalo de tempo cada vez mais curto para continuar sendo competitivas.

Ao mesmo tempo, a funcionalidade que se espera de um produto e, consequentemente, a complexidade envolvida na sua elaboração, tem aumentado bem rápido. Um exemplo são os telefones celulares que agora oferecem muitas funções além da comunicação básica, como acesso à Internet, organizador pessoal e jogos.

A tecnologia de fabricação atual permite a utilização de mais de 200 milhões de transistores [54] por *chip* porém, os processos de desenvolvimento usuais dos *system-on-chip* (SoC) não conseguem explorar todo o potencial oferecido pela tecnologia.

Os primeiros SoCs não eram complexos, normalmente possuíam uma ou duas unidades de processamento, por exemplo um microcontrolador e um processador digital de sinais, além de algum *hardware* adicional, periféricos e memória. Já os projetos atuais de SoC contam cada vez mais com unidades de processamento para substituir blocos lógicos

complexos e com isso reduzir o tempo de desenvolvimento.

2.4.1 Hydra: uma arquitetura com múltiplos processadores

Alguns estudos foram feitos focando arquiteturas específicas, fixas, de múltiplos processadores em um chip, um destes projetos é o Hydra [55], da Universidade Stanford.

O Hydra foi elaborado para servir como estudo do conceito de múltiplos processadores em um chip. A arquitetura utilizada é fixada como quatro processadores superescalares MIPS em um *chip*, cada um deles similar a um R10000 com um nível de cache de instruções e de dados integrado. Para compilar programas para esta arquiteturas utiliza-se o sistema de compilação SUIF [56], também desenvolvido em Stanford.

Esta arquitetura é focada em desempenho do acesso à memória, o que é interessante para diversas aplicações atuais com demanda crescente, como as de multimídia, que manipulam uma grande quantidade de dados. Porém, uma desvantagem em se utilizar uma arquitetura fixa é que nem todas as aplicações têm o mesmo requisito. A arquitetura utilizada pelo Hydra é certamente muito boa, porém pode ser muito cara para o orçamento de muitos sistemas.

2.4.2 FITS: Refinando o conjunto de instruções

Vários projetos poderiam se aproveitar de processadores menos potentes (e bem mais baratos) que poderiam ser refinados para que contivessem um conjunto de instruções reduzido. Esta técnica possibilita que cada processador tenha somente o conjunto de instruções necessário para resolver a sua parte do problema.

Allen et. al., através do projeto FITS [57] da Universidade de Michigan, fizeram um estudo das instruções mais utilizadas pelo *benchmark* MiBench [58] e detectaram que apenas 20 instruções do ARM [59] são suficientes para cobrir 95% de toda execução dos programas. Constata-se que, em média, 55,6% das instruções são usadas somente em

1% do tempo de execução dos programas. Portanto, instruções executadas com pouca frequência podem ser transformadas em software sem afetar o desempenho de forma significativa. A redução do número total de instruções traz algumas vantagens, por exemplo a lógica de decodificação é simplificada, permitindo economizar área e tornar o processador mais barato.

Esta idéia é interessante, mas devemos lembrar que os programas levados em conta no estudo foram compilados a partir de uma linguagem de alto nível e não feitos manualmente em *assembly* nativo. Não existe garantias que um compilador utilize de forma eficiente o conjunto de instruções. Na maioria dos casos, podemos culpar o compilador por não se utilizar da maioria das instruções disponíveis. Com o avanço dos estudos na área de compiladores, é possível que cada vez mais instruções diferentes sejam utilizadas.

Os membros do projeto FITS calcularam que, apesar das instruções de máquinas RISC possuirem três operandos, em 60% dos casos apenas dois operandos são necessários. Portanto é possível economizar no tamanho das instruções intermeando instruções de dois operandos e de três operandos. Calcularam ainda que a quantidade de operandos imediatos únicos (sem contar repetições) utilizados na prática é muito menor que o tamanho reservado a eles no campo das instruções. Este fato pode levar a mais uma economia de espaço se os imediatos forem armazenados em uma tabela e as instruções contarem apenas com índices para esta tabela.

A tendência de processadores pequenos e simples substituindo blocos lógicos parece ter vindo para ficar e deve culminar com centenas de processadores heterogêneos dentro de um SoC conectados através de uma rede interna à pastilha (*network-on-chip*, NoC [60]).

2.4.3 SUNMAP: geração automática de NoCs

Uma fase importante no desenvolvimento de NoCs é o mapeamento dos núcleos processadores (*cores*) em uma topologia que case com as necessidades da aplicação. A ferramenta

SUNMAP [61], da Universidade Stanford se propõe a automatizar este passo, levando em conta alguns objetivos como minimizar o tempo de comunicação, área e potência dissipada.

A ferramenta leva em conta 5 topologias homogênias: grade, torus, hipercubo, *clos* e borboleta; mas comenta que outras topologias complexas como octagonal ou estrela podem ser incorporadas à biblioteca de topologias.

Para escolha da topologia, usa-se uma simulação funcional. Só após a topologia ser escolhida é que se gera um modelo SystemC que pode ser simulado com precisão de ciclos.

2.4.4 Sistemas configuráveis com ADLs

Para reduzir o tempo de desenvolvimento da arquitetura multi-processada, abordagens que utilizam ADLs são promissoras, pois permitem a exploração e otimização conjunta dos dois aspectos referidos nas seções anteriores: além de poder testar um conjunto de instruções específico para cada processador, possibilita também que várias topologias e interconexões de NoCs possam ser testadas.

Para aplicação dessas idéias de otimização global, o projeto de SoCs deve ser feito levando em conta todo o sistema. A Universidade de Aachen, usando a infra-estrutura do projeto LISA, criou uma metodologia [62] que facilita o desenvolvimento de um sistema otimizado e o seu teste de corretude. A comunicação é projetada no nível de transação (TLM), utilizando interfaces padronizadas para o SystemC [63].

A metodologia utiliza uma abordagem de refinamento conjunta para o processador e para o sistema de comunicação. Isso permite que sejam levados em conta os aspectos de eficiência, precisão e desempenho a cada nível de abstração.

O projeto ArchC está passando por uma reestruturação para que sistemas com vários processadores possam ser simulados. A comunicação entre os processadores do sistema utilizará também a abordagem por transações, que permite um desempenho alto de si-

mulação. O Capítulo 5 traz informações detalhadas sobre a criação de sistemas utilizando ArchC.

Capítulo 3

Emulação de Chamadas de Sistema para Simuladores

O aumento de complexidade das aplicações atuais, notadamente aplicações multimídia em dispositivos embarcados, aumenta a necessidade de armazenamento e manipulação de dados para estas plataformas. A simulação destes sistemas complexos sem o uso de um sistema de arquivos certamente se torna um pesadelo. Mesmo dispositivos com fortes restrições, como *handhelds* e *palmtops*, estão gradualmente incorporando memórias não-voláteis que permitem implementações de sistemas de arquivos.

Neste cenário, modelos que simulam todo o sistema e provêem suporte a sistemas operacionais se tornam ferramentas essenciais para análise de funcionalidade e desempenho. Alguns trabalhos na literatura são capazes de produzir simuladores a partir de modelos escritos em linguagens de descrição de arquiteturas (ADLs) como EXPRESSION [35], LISA [37], nML [64], ISDL [22] e outras. Apesar de sistemas operacionais de tempo real (RTOS) estarem na moda para os sistemas embarcados, ainda não existem ADLs que gerem simuladores com suporte a chamadas do sistema operacional. Normalmente se toma uma abordagem simplificada para solução deste problema, que é incorporar ao programa simulado todos os dados de entrada e saída como vetores estáticos. Infelizmente, esta abordagem requer que o programa seja recompilado cada vez que se deseja uma mudança

no conjunto de dados.

Este capítulo propõe uma metodologia para possibilitar que simuladores de arquiteturas utilizem chamadas do sistema operacional. A idéia proposta é genérica para ser aplicada a qualquer simulador de conjunto de instruções.

O objetivo foi prover suporte a entrada e saída para arquivos e alocação dinâmica de memória. Para isso, foi disponibilizada para a aplicação simulada rotinas de sistema compatíveis com o padrão POSIX, portanto o código fonte da aplicação não sofre nenhuma modificação. As operações de entrada e saída são realizadas de forma transparente. Leituras e escritas para arquivos são repassadas para o sistema de arquivos da máquina hospedeira, o repasse também ocorre dos acessos ao console virtual para o real (teclado e tela).

Esta metodologia, que é uma das contribuições deste trabalho de doutorado, foi aplicada com sucesso nos simuladores do projeto ArchC, tanto o simulador compilado quanto o interpretado (com precisão de instruções e precisão de ciclos). Os simuladores gerados para as arquiteturas MIPS I e SPARC V8 foram avaliados através da execução de programas dos *benchmarks* Mibench e Mediabench.

O restante deste capítulo contém na Seção 3.1 alguns trabalhos já realizados na emulação de sistemas operacionais por simuladores e na Seção 3.2 a abordagem genérica para solução do problema. Os resultados dos experimentos estão na Seção 3.3. Finalmente a Seção 3.4 conclui o capítulo.

3.1 Trabalhos Relacionados

Não existe informação suficiente na literatura para se afirmar que os simuladores gerados com ADLs suportam emulação de sistema operacional. A ADL EXPRESSION, que abriu o código de algumas ferramentas recentemente, gera simuladores sem suporte a entrada e

saída.

Simplescalar [65] é um simulador de conjunto de instruções bem conhecido mas que limita o redirecionamento a arquiteturas do tipo MIPS. Ele emula chamadas ao sistema operacional pelo mecanismo disponível no conjunto de instruções do MIPS. Uma chamada de sistema é iniciada pela instrução `SYSCALL`, mas antes alguns parâmetros são inicializados. O registrador `$v0` deve ser carregado com o código da chamada de sistema e argumentos adicionais, de acordo com a rotina selecionada, devem ser passados através dos registradores `$a0-$a3`. Este mecanismo não é genérico o suficiente, pois conta que o conjunto de instruções possua instruções especificas para chamadas de sistema.

O simulador Shade [66], da SUN, não é redirecionável, embora possua versões para arquiteturas MIPS I, SPARC V8 e SPARC V9. Ele usa uma técnica de compilação dinâmica para atingir um bom desempenho. O principal propósito do Shade é instrumentar o código e extrair o número de ciclos simulados. Chamadas de sistema são suportadas, mas as arquiteturas alvo são escritas manualmente e devem ser similares às arquiteturas RISC.

O projeto GNU possui um simulador que pertence ao pacote GNU Debug (GDB[1]). Ele é acessado a partir do depurador com o comando `target sim`. Este simulador possui suporte a chamadas de sistema, porém, ele está disponível para um conjunto relativamente pequeno de arquiteturas, já que ele não é suportado por todas as arquiteturas que o GDB pode ser redirecionado. Uma funcionalidade importante, que não é suportada, é a capacidade de obter argumentos da linha de comando. Programas que requerem estes argumentos devem ser modificados para que os argumentos sejam codificados manualmente, e de maneira fixa, no programa.

Não foi possível verificar, pela informação disponível na literatura sobre as ADLs como LISA [37], nML [64], ISDL [22] e outras, se estas linguagens possuem qualquer mecanismo

[1]GNU Debug homepage: `http://www.gnu.org/software/gdb`

para permitir a emulação de chamadas de sistema.

Apesar deste trabalho estar baseado em simuladores gerados automaticamente pela ADL ArchC [40], as idéias expostas podem ser implementadas em qualquer simulador.

3.2 Uma Técnica Flexível de Emulação de S.O.

Quando novos modelos de arquiteturas são implementados em ArchC, alguns testes são feitos para verificar sua corretude. Nesta fase de testes as instruções são analisadas. Este é um trabalho necessário, porém cansativo e pode ser difícil testar todas as condições possíveis de uso de cada instrução. É mais interessante testar as funcionalidades da arquitetura com exemplos de instruções presentes em programas reais. Quanto maior e mais complexo for o programa, melhor é o uso que eles fazem do conjunto de instruções e melhor fica a cobertura dos testes.

A maioria dos simuladores descritos na Seção 3.1 não consegue manipular chamadas de sistema corretamente. O objetivo maior deles é simular o conjunto de instruções da arquitetura alvo, porém eles não oferecem recursos para simulação de programas complexos. Aplicações embarcadas lidam com grande quantidade de dados de entrada e, se o simulador não for capaz de lidar com entrada e saída, a entrada não pode ser provida dinamicamente para o programa como se espera em uma execução real. A abordagem usual para este problema é alocar um vetor grande para guardar os dados fixos de entrada e outro para receber a saída processada. Isso significa que a corretude do programa só pode ser verificada pela comparação dos *traces* gerados pelo simulador com os *traces* gerados pelo hardware real ou usando outro vetor que inclua a saída esperada e compará-la com a saída da execução (o que requer mais mudanças no código original).

Um dos objetivos deste trabalho foi obter uma maneira genérica de acoplar as funcionalidades de um sistema de E/S e de um sistema de arquivos em qualquer simulador de

conjunto de instruções. Isso facilita a verificação do sistema, já que a aplicação simulada pode ler seus dados de entrada diretamente de um arquivo, ou mesmo do teclado, e escrever sua saída para outro arquivo. As mensagens de erro e de depuração podem ser direcionadas para a tela da máquina hospedeira para que erros sejam encontrados rapidamente. É importante lembrar que os códigos fonte dos *benchmarks* não precisam de nenhuma modificação, enquanto que opções adicionais para o programa simulado podem ser dadas pela linha de comando (que serão copiadas da linha de comando do simulador).

Quando programas maiores foram executados, na tentativa de uma maior cobertura de instruções, notou-se que alguns *benchmarks* precisam não só de um sistema de arquivo para serem executados, mas também requerem alocação de memória dinâmica e funções que lidam com temporização, por exemplo. Estas funções extras foram incluídas no simulador e esta funcionalidade foi denominada Emulação de Sistema Operacional.

3.2.1 Metodologia

Em algumas placas de prototipagem, a funcionalidade de entrada e saída é implementada através da porta serial destes dispositivos, que usam o computador como um terminal. Quando um programa requer ou emite um dado, este dado é repassado ao computador conectado à placa e este pode ler do teclado ou exibir na tela, de acordo com a requisição, e indicar de volta ao programa na placa que o dado foi exibido ou entregar o dado lido.

Infelizmente, esta idéia só funciona para sistemas com dois processadores, como o descrito acima, porque o processador da placa de prototipagem, após fazer uma requisição para o computador terminal, fica parado aguardando uma resposta. Já o computador terminal executa um processo cuja única função é servir as requisições do processador da placa. Ele aguarda um pedido, o executa e volta a aguardar outro pedido. Para que este sistema de requisições seja implementado em um simulador, são necessários dois processos: o simulador propriamente dito e o emulador de terminal que serve as requisições

de entrada e saída.

O uso de processos concorrentes, que é uma técnica de programação um pouco mais complicada, foi evitado. Foi utilizada a idéia encontrada em alguns simuladores de designar endereços especiais da memória de programa da máquina simulada para troca de dados com o sistema da máquina hospedeira.

Observe que esta emulação requer que o sistema da máquina hospedeira também seja considerado, já que é ele quem vai realizar as operações requisitadas. Portanto, alguma interface de comunicação entre a chamada de S.O. da aplicação e o S.O. nativo é necessária.

A Figura 3.1 mostra os níveis de abstração considerados na metodologia proposta. A modelagem da máquina hospedeira usa três níveis de abstração. O nível mais baixo é o de hardware, o processador que executa o fluxo de instruções. O próximo nível é o de S.O., que provê algumas funcionalidades usadas nos programas, como E/S, sistema de arquivos e gerenciamento de memória. O último nível é a aplicação, onde se encontra o programa do usuário. O simulador roda como uma aplicação na máquina hospedeira e simula os dois níveis mais baixos da arquitetura alvo.

O programa a ser simulado é compilado por um cross-compilador para a arquitetura alvo e o binário resultante serve como entrada para o simulador, complementando o nível de aplicação para a máquina alvo virtual. Estes níveis de abstração facilitam as mudanças no processador e sistema operacional usados no ambiente de execução.

As chamadas de sistema são implementadas reservando-se um bloco de endereços da memória de programa e associando um endereço para cada rotina do sistema que precise uma interação com o sistema nativo. Não são todas as rotinas que precisam desta interação por causa de algumas incompatibilidades, como veremos a seguir. Esta abordagem é bem genérica e pode ser implementada em outros simuladores.

Um salto do programa para algum dos endereços especiais reservados deve se comportar como uma chamada de sistema. Neste caso, são necessárias informações adicionais

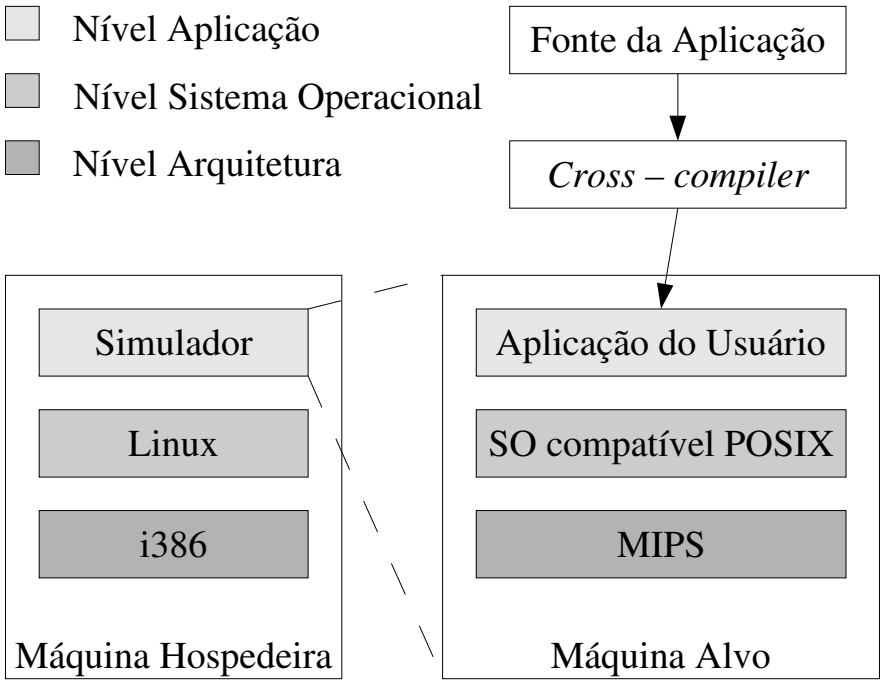


Figura 3.1: Níveis de Abstração

sobre comportamento da arquitetura alvo para realizar três operações: (1) passar os argumentos para a rotina do sistema; (2) passar o valor de retorno da rotina de volta para o programa; (3) retornar a execução do programa do ponto imediatamente após a chamada da rotina. Estas informações são normalmente encontradas no manual *Application Binary Interface* (ABI) [5, 67] da arquitetura, que especifica, entre outras informações, como é a chamada de procedimentos, inclusive fixando os registradores e outros recursos que são usados. As próximas seções explicam as mudanças feitas em cada bloco da Figura 3.1 para resolver os principais problemas associados a esta implementação.

3.2.2 Modelando a Camada de Abstração do S.O.

A programação em uma linguagem de alto nível como C/C++ é uma necessidade para o desenvolvimento atual. A maioria das aplicações são desenvolvidas em C/C++ com

alguns trechos em *assembly* para melhora do desempenho de laços internos críticos.

Qualquer programa não trivial em C/C++ usa a biblioteca padrão de C, que provê rotinas essenciais de E/S, alocação de memória e outras utilidades. A implementação mais usada desta biblioteca nos sistemas Linux é a GNU C Library, conhecida como `glibc`. Porém, a implementação mais apropriada para sistemas embarcados, por sua preocupação de tamanho do código, é a Cygnus Newlib.

A biblioteca Newlib implementa rotinas de interação com o sistema, como entrada e saída (de/para terminal ou arquivos), gerenciamento de memória, temporização, etc. Muitas destas rotinas não chamam diretamente o sistema, ao invés disso usam um pequeno conjunto de funções para esta tarefa. Por exemplo, todas as rotinas de entrada e saída em C, como a família `printf()`, `fwrite()`, etc, usam apenas 4 chamadas de sistema: `open()`, `close()`, `read()`, `write()`.

Esta funcionalidade não foi implementada com interrupções de *software* porque cada arquitetura pode tratar interrupções de um modo particular. A metodologia usada é similar ao que ocorre em alguns simuladores de placas de protótipos. Uma faixa de endereços é reservada de modo que uma tentativa de executar uma instrução nesta faixa faz com que o simulador execute um comportamento especial associado ao endereço específico. O simulador chama a rotina de sistema associada a este endereço. Portanto, uma instrução real que tenha sido carregada junto com o programa para este endereço perde o seu significado, no sentido que ela não será executada. Os cross-compiladores de programas que serão simulados em ArchC foram configurados para desconsiderar a faixa de endereço de 0x40 a 0x100 bytes na ligação. Estes endereços são preenchidos com zeros nos binários.

Os argumentos passados para a rotina de sistema simulada devem ser convertidos para argumentos para a rotina real que será executada na máquina hospedeira. O modo que isso é feito varia de um processador para o outro, como descrito no manual da ABI da arquitetura.

Como desenvolvedor de uma nova arquitetura, o projetista é capaz de informar quais elementos de armazenamento (memória, banco de registradores, etc) guardam os argumentos de uma chamada de função. O conjunto de informações necessário para que a ADL gere uma interface para as rotinas de sistema são:

- Como obter os primeiros três argumentos de uma chamada de função e como distinguir o tipo do argumento, que pode ser um número inteiro, um ponteiro ou mesmo uma cadeia de caracteres (em alguns casos uma conversão de *endian* é necessária);
- Como e onde (memória ou registrador?) salvar o valor de retorno da função como o tipo inteiro ou ponteiro (também pode requerer conversão);
- Como retornar o fluxo de controle para a próxima instrução do programa. Normalmente usa-se um salto para o endereço de retorno gravado em um registrador específico para este fim (mas qual é este registrador?);
- Como armazenar os argumentos da linha de comandos para que eles possam ser acessados na rotina `main()` do programa simulado;

Estas são as informações dependentes de arquitetura. Toda funcionalidade de sistema operacional é implementada como uma nova classe em ArchC denominada `ac_syscall`. Ela é uma classe genérica, independente da arquitetura, mas possui métodos virtuais que precisam ser especializados para cada novo processador de modo a refletir a descrição da ABI, que é a parte dependente da arquitetura.

3.2.3 Exemplo: função `open()`

A rotina de sistema `open()` é usada para exemplificar a emulação de sistemas. Outras rotinas são implementadas de maneira similar. O protótipo no padrão POSIX para esta função é:

```
int open(const char *pathname, int flags, mode_t mode);
```

Foi implementada uma função no simulador que faz a interface entre a chamada do programa simulado com a chamada do sistema nativo para a `open()`, apresentada na Figura 3.2. Note que esta função é independente da arquitetura mas usa algumas funções-interface especializadas, como a `get_buffer()` e `get_int()`. As funções-interface são descritas pelo projetista do novo processador (Seção 3.2.4).

```
1  void ac_syscall::open()
2  {
3    unsigned char pathname[100];
4    get_buffer(0, pathname, 100);
5    int flags = get_int(1); correct_flags(&flags);
6    int mode = get_int(2);
7    int ret = ::open((char*)pathname, flags, mode);
8    if (ret == -1) {
9      RUN_ERR("System Call open (file %s): %s\n",
10             pathname, strerror(errno));
11     exit(EXIT_FAILURE);
12   }
13   set_int(0, ret);
14   return_from_syscall();
15 }
```

Figura 3.2: Função de interface para rotina `open()`

Conforme pode ser visto no protótipo da função, o primeiro argumento para `open()` é uma *string* representando o nome do arquivo que deve ser aberto. A função `get_buffer()` é usada para coletar esta *string*. Ela é responsável por: encontrar o primeiro argumento, interpretá-lo como um ponteiro para caractere, ler a memória do processador virtual, converter a ordem dos bytes (*endian*) quando necessário e retornar a *string* como um ponteiro para caractere no espaço de endereçamento real da máquina hospedeira.

O próximo argumento da `open()` é um número inteiro que codifica alguns bits de opções. Este argumento é resgatado do sistema simulado através da chamada `get_int(1)`,

que é capaz de encontrar o argumento `1` (que é o segundo, pois a contagem começa em zero) de maneira dependente da ABI do processador simulado e retorná-lo como um inteiro. O terceiro argumento também é um número inteiro e é tratado de maneira similar.

Todos os três argumentos são coletados da aplicação rodando no simulador. Em seguida, eles são passados para a execução da função `open()` nativa. O valor de retorno, que é um inteiro, precisa ser devolvido para a aplicação. A função `set_int()` toma as medidas necessárias para colocar este valor no lugar esperado.

Existe ainda um outro valor que deve ser comunicado para a aplicação. Ele é a variável global `errno`, usada por todas as funções da biblioteca C padrão como um código de erro complementar. O problema é que cada aplicação simulada vai alocar esta variável em uma posição diferente de memória de dados (isso depende do ligador) e não existe um meio simples de saber qual é esta posição se a aplicação carregada no simulador não possuir informação de símbolos, a mesma usada para depuração. Como o simulador não pode avisar a aplicação que ocorreu algum erro, escolhemos tratar os erros no próprio simulador, mostrando a mensagem de erro e abortando a simulação com um código de erro, como exemplificado na Figura 3.2. Esta abordagem não prejudica os programas bem comportados, como todos os dos *benchmarks* Mediabench e MiBench, pois suas rotinas de sistemas sempre executam corretamente.

3.2.4 Interface Binária da Aplicação (ABI)

Funções que sabem como coletar os parâmetros das chamadas de sistema são um exemplo das funções-interface que precisam ser implementadas para cada arquitetura alvo modelada em ArchC. Estas funções são normalmente muito pequenas, com menos de 5 linhas de código. Esta informação pode ser encontrada diretamente no manual da ABI. A Figura 3.3 mostra como exemplo algumas destas funções para a arquitetura MIPS e como elas são implementadas usando ArchC para acessar o banco de registradores e a memória.

Estas funções codificam quais registradores são usados para passagem de parâmetros, onde é armazenado o valor de retorno e como retornar ao fluxo ao programa.

O processador MIPS I será usado para demonstrar como são as sub-rotinas dependentes de arquitetura. Neste processador, os registradores de 4 a 7 guardam os primeiros 4 argumentos de uma chamada de função e o retorno fica no registrador 2. As chamadas de sistema utilizam normalmente no máximo 3 argumentos, por isso, o acesso além dos 4 argumentos iniciais foi desconsiderado. Os objetos RB e MEM representam o banco de registradores e a memória, respectivamente. A Figura 3.3 mostra algumas sub-rotinas de tratamento de argumentos utilizadas no MIPS I.

A rotina `get_int()` lê um número inteiro passado como argumento e a `set_int()` escreve um valor de retorno como um número inteiro. As funções `get_buffer()` e `set_buffer()` são usadas de forma similar, porém para o tipo de dados *string*. A `set_prog_args()` é usada para passar os argumentos de linha de comando para o programa simulado e só é chamada no início da simulação. Por fim a `return_from_syscall()` é chamada para retornar o fluxo de operação para o programa simulado, fazendo o contador de programa apontar para instrução seguinte à chamada de sistema.

As funções apresentadas na Figura 3.3 são especializações dos métodos da classe base de chamada de sistema do ArchC chamada `ac_syscall`, que tem 265 linhas de código e não precisa ser modificada, pois é independente de arquitetura (um exemplo é a `open()` mostrada em Figura 3.2). São 7 os métodos que devem ser especializados para cada arquitetura. A Figura 3.3 mostra 4 destes métodos para arquitetura MIPS I, que no total foram implementados com somente 51 linhas de código. Para o SPARC V8, o mesmo número de linhas foram necessárias, já que a diferença entre estas duas arquiteturas em relação a ABI é somente a numeração dos registradores de argumentos e o de retorno. A pequena quantidade de linhas nestes arquivos mostra como é simples a adaptação destes métodos para novas arquiteturas.

```
1  void mips1_syscall::get_buffer(int argn,
2                                 unsigned char* buf,
3                                 unsigned int size)
4  {
5    unsigned int addr = RB.read(4+argn);
6
7    for (unsigned int i = 0; i<size; i++, addr++) {
8      buf[i] = MEM.read_byte(addr);
9    }
10 }
11
12 int mips1_syscall::get_int(int argn)
13 {
14    return RB.read(4+argn);
15 }
16
17 void mips1_syscall::set_int(int argn, int val)
18 {
19    RB.write(2+argn, val);
20 }
21
22 void mips1_syscall::return_from_syscall()
23 {
24   ac_pc = RB.read(31);
25 }
```

Figura 3.3: Algumas funções de interface para arquitetura MIPS

3.2.5 Requisitos do Compilador

Conforme já mencionado, as chamadas de sistema para arquitetura alvo são mapeadas em um bloco de memória reservado onde cada endereço é associado a uma chamada de sistema. Isso requer uma ação do ligador.

O GNU Compiler Collection (GCC) foi escolhido como o compilador para os testes pela sua facilidade de configuração e disponibilidade em várias plataformas. Essa seção descreve as mudanças na configuração do GCC para implementar a abordagem de chamadas de sistemas em um cross-compilador GCC[2]:

[2]CrossGCC FAQ: `http://www.objsw.com/CrossGCC`

Criar um novo arquivo `spec`. O arquivo `spec` é a principal forma de configuração do GCC. Se um novo arquivo de configuração for especificado na linha de comando, o conteúdo desse é mesclado com o conteúdo da configuração original. Portanto, apenas pequenas mudanças na configuração original precisam estar no novo arquivo `spec`. A Figura 3.4 mostra estas mudanças: um novo arquivo de início (`*startfile:`), nenhum arquivo de fim (`*endfile:`), um novo arquivo de *script* para o ligador (`*link:`) e o uso da biblioteca de chamadas de sistema do ArchC (`-lac_sysc` em `*lib:`).

```
1  *link:
2  -L/l/archc/compilers/ac_specs/mips1 -Tac_link.ld
3
4  *startfile:
5  ac_start.o
6
7  *endfile:
8
9
10 *lib:
11 -lc -lac_sysc
```

Figura 3.4: Novo arquivo de *spec* para o GCC

Criação de um novo arquivo de início. Este arquivo contém uma rotina de inicialização em *assembly* dependente da arquitetura. O registrador de pilha e alguns outros registradores podem ser inicializados aqui. Este é o ponto de entrada da simulação.

Criação do arquivo de ligação. O *script* de ligação informa ao ligador como unir em um binário final os vários arquivos objeto. Neste arquivo são especificados o endereço de entrada (escolhido o endereço 0 por simplicidade) e indica-se que o arquivo de início é seguido por um bloco de endereços reservados para as chamadas de sistema.

3.2.6 Requisitos do Simulador

O simulador também deve sofrer pequenas mudanças para que o bloco de endereços reservados seja tratado de forma especial. Os próximos parágrafos informam as modificações nos simuladores.

Repasse das opções de linha de comando. Antes de iniciar a simulação, uma função de interface é chamada para iniciar os argumentos que serão passados para a função `main()` da aplicação simulada. Estes argumentos são copiados diretamente da linha de comando do simulador, após remoção das opções específicas do simulador.

Reconhecimento de chamadas de sistema. Quando o fluxo de execução atinge um endereço reservado, ao invés de passar pelo ciclo natural de busca, decodificação e execução da instrução, o simulador deve chamar uma função especial que corresponde a uma chamada de sistema associada àquele endereço. Esta função toma os seguintes passos:

- Coletar os argumentos necessários, fazendo as conversões necessárias (por exemplo *endian*);
- Executar a chamada de sistema da máquina nativa;
- Retornar o resultado para a aplicação simulada, novamente fazendo conversões se necessário;
- Retornar para o ponto da aplicação imediatamente seguinte à chamada de sistema;

Esta abordagem foi testada usando o simulador compilado gerado pelo ArchC para as descrições de MIPS I e SPARC V8. As mudanças no simulador compilado foram mais simples que no interpretado, já que os estágios de busca e decodificação são pré-calculados para o simulador compilado. Foi necessário somente incluir as chamadas das funções especiais nas respectivas posições do programa C++ gerado.

Repassar o código de saída da aplicação para o simulador. Após o fim da simulação, o simulador chama a rotina de interface `get_exitcode()` para capturar o código de saída da aplicação e acaba sua atividade passando este mesmo código como o seu código de saída.

A metodologia apresentada funciona para simuladores de conjunto de instruções. Para usar as mesmas funções em simuladores com precisão de ciclos, o simulador deve primeiro esvaziar o *pipeline* normalmente, impedindo a entrada de novas instruções. Com o *pipeline* vazio, pode-se garantir que os registradores possuem dados consistentes e que as funções de chamadas de sistema coletem os seus argumentos de forma correta, sem usar os recursos de *bypassing* e *forwarding* do *pipeline*.

3.2.7 Biblioteca de Rotinas de Sistema

As rotinas de chamadas ao sistema foram agrupadas em uma biblioteca escrita na linguagem C, de forma que apenas uma recompilação é necessária para que ela funcione com uma nova arquitetura alvo. As rotinas que requerem interação com o sistema e, portanto, possuem um endereço reservado correspondente, são implementadas em C por um comando `goto` para este endereço reservado. A biblioteca deve ser usada durante a compilação dos programas que serão rodados no simulador.

A Tabela 3.1 mostra as rotinas de sistema que foram implementadas até o momento. As rotinas que interagem com o sistema nativo estão marcadas na terceira coluna. Para facilitar a leitura, a tabela agrupa as rotinas de acordo com sua funcionalidade: Entrada e Saída, Controle, Temporização e Alocação de Memória. Nem todas as rotinas podem interagir com o sistema nativo. Isso se deve ao fato que a implementação da biblioteca de C usada no simulador (Newlib) possui uma incompatibilidade com a implementação da usada nativamente (GLibC) em relação à algumas estruturas de dados. As rotinas que interagem corretamente com o sistema são somente as que trabalham com tipos de dados

Grupo	Função	Interação com o sistema nativo
E/S	open	√
	creat	√
	close	√
	read	√
	write	√
	isatty	√
	lseek	√
	fstat	√
Controle	_exit	√
	chmod	
	chown	
	stat	
	getpid	
	kill	
	unlink	
Temporização	time	√
	times	
	gettimeofday	
Memória	sbrk	√

Tabela 3.1: Rotinas de sistema já implementadas

que têm o tamanho de uma palavra ou com *strings*.

A Figura 3.5 mostra a abordagem proposta, resumindo como o fluxo de controle é transferido do programa do usuário executado dentro do simulador ArchC para uma rotina do S.O. nativo. Primeiro, o programa do usuário chama normalmente a função `fopen` original da Newlib. A `fopen` possui uma chamada para função `open`, que é definida pela Biblioteca de Syscalls do ArchC, que foi ligada junto com o programa do usuário. Depois de chamada, a `open` acessa um endereço reservado e isso faz com que o simulador perceba que a funcionalidade de abertura de arquivos está sendo requisitada. O simulador, neste ponto, usa as funções de interface apropriadas para coletar os argumentos da função `open` e repassá-los para a `open` nativa. Enfim, a rotina `open` nativa consegue realizar a tarefa pedida.

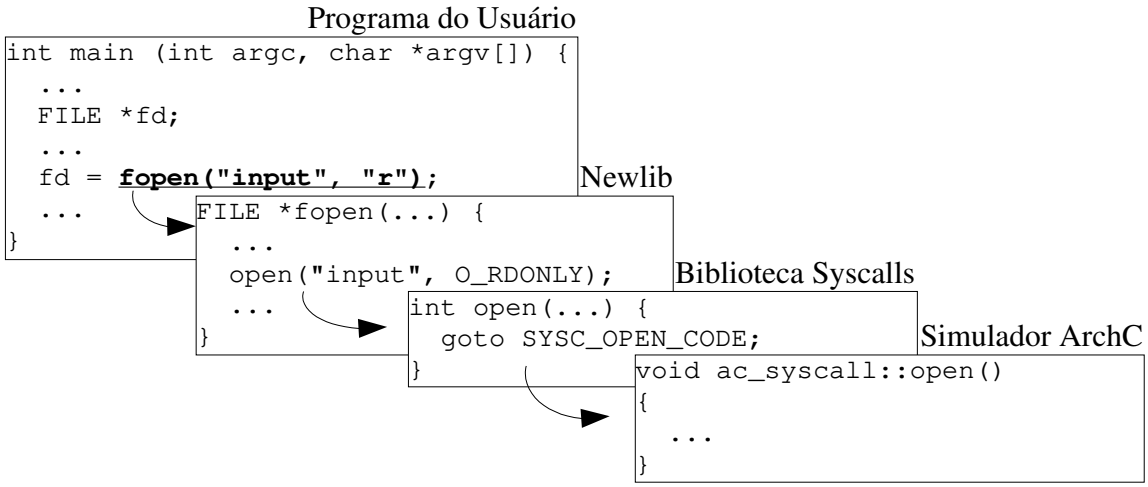


Figura 3.5: Sequência para chamar o S.O. nativo

3.3 Experimentos

Essa metodologia foi aplicada nas ferramentas de geração de simuladores da ADL ArchC. Foram gerados simuladores com suporte a chamadas de S.O. a partir do modelo MIPS I, nos níveis de precisão de instrução e de ciclos, e do modelo SPARC V8, no nível de precisão de instrução[3]. Como ambos os níveis de abstração do MIPS I simulam o mesmo conjunto de instruções, é possível executar o mesmo binário do MIPS I nos dois simuladores MIPS I gerados.

Foi feito um experimento instrumentando os simuladores MIPS I e SPARC V8 para contar quantas vezes as rotinas de sistema foram chamadas. O resultado deste experimento para o MIPS I e para o SPARC V8 pode ser conferido na Tabela 3.2 e na Tabela 3.3, respectivamente. Os experimentos foram realizados em um Pentium 4 de 2.4GHz com sistema Linux. Nesta configuração, o simulador é capaz de executar uma média de 120 milhões de instruções por segundo.

As primeiras duas colunas da Tabela 3.2 e da Tabela 3.3 mostram alguns programas

[3]Um modelo com precisão de ciclos para o SPARC V8 está em desenvolvimento

executados e o *benchmark* de onde vieram. Usamos dois *benchmarks*: Mediabench [68] e Mibench [58]. Os programas do Mibench são divididos em duas versões que processam volume de dados diferentes. Nessas tabelas, o caractere sobrescrito foi usado para distinguir entre as versões *small* e *large* e o parênteses para marcar quando o programa fonte é o mesmo e somente as entradas variam. Alguns programas, como o `pegwit`, requerem parâmetros de linha de comando que também são colocados entre parênteses (o `pegwit` foi executado duas vezes, uma com a opção de cifrar e outra de decifrar).

Benchmark	Programa	Tam. (instr.)	open	close	read	write	lseek	fstat	sbrk	Instruções executadas
Mediabench	adpcm rawcaudio	7,008			149	149				7,491,406
	adpcm rawdaudio	7,001			149	149				5,902,037
	jpeg cjpeg	33,896	2	2	100	6		2	12	16,776,931
	jpeg djpeg	36,003	2	2	7	100		2	6	5,902,037
	mpeg2 mpeg2encode	32,685	15	15	503	161		15	24	11,699,579,115
	mpeg2 mpeg2decode	22,483	13	13	20	144	2		12	3,858,000,780
	pegwit (encrypt)	21,482	3		186	90	2	4	5	31,027,254
	pegwit (decrypt)	21,482	2		107	90	1	2	4	17,497,043
Mibench	basicmath_small	13,839				19,733		1	2	1,386,360,829
	basicmath_large	14,057				492,999		1	2	22,469,864,764
	bitcount (small)	9,551				12		1	2	45,593,411
	bitcount (large)	9,551				12		1	2	684,250,119
	qsort_small	12,330	1		54	10,003		2	3	14,359,301
	qsort_large	14,435	1		1,537	50,003		2	2	991,774,387
	susan (smooth. small)	18,417	2	2	8	8		2	5	35,318,139
	susan (smooth. large)	18,417	2	2	109	109		2	5	423,335,580
	dijkstra_small	12,190	1		29	222		2	4	59,276,836
	dijkstra_large	12,190	1		29	1,177		2	4	284,841,999
	crc (small)	9,165	1	1	1,338	1		2	2	31,642,843
	crc (large)	9,165	1		25,989	1		2	2	588,282,883
	fft (small)	11,896				116		1	5	760,568,610
	fft (large)	11,896				952		1	5	15,646,419,915
	sha (small)	7,556	1	1	306	1		2	2	13,036,286
	sha (large)	7,556	1	1	3,173	1		2	2	137,088,164
	string search_small	8,528				57		1	2	275,212
	string search_large	8,524				1,332		1	2	6,844,944

Tabela 3.2: Contagens de chamadas de sistema para o MIPS I

As próximas colunas da Tabela 3.2 e da Tabela 3.3 mostram o número de vezes que cada rotina de sistema foi chamada durante a execução do programa. Células vazias indicam que a rotina não foi chamada pelo programa correspondente. O programa

Benchmark	Programa	Tam. (instr.)	open	close	read	write	lseek	fstat	sbrk	Instruções executadas
Mediabench	adpcm rawcaudio	6,911			149	149				7,256,632
	adpcm rawdaudio	6,906			149	149				6,261,168
	jpeg cjpeg	24,906	2	2	100	6		2	12	14,288,756
	jpeg djpeg	29,638	2	2	7	100		2	5	4,187,862
	mpeg2 mpeg2encode	29,438	15	15	503	161		15	24	10,834,241,285
	mpeg2 mpeg2decode	21,113	13	13	20	144	2		12	3,451,835,100
	pegwit (encrypt)	19,590	3		186	90	2	4	5	30,897,936
	pegwit (decrypt)	19,590	2		107	90	1	2	4	16,863,601
Mibench	basicmath_small	12,658				19,733		1	2	1,305,099,573
	basicmath_large	12,825				492,999		1	2	21,365,365,697
	bitcount (small)	9,159				12		1	2	49,862,749
	bitcount (large)	9,159				12		1	2	748,368,498
	qsort_small	11,782	1		54	10,003		2	3	14,137,667
	qsort_large	13,549	1		1,537	50,003		2	3	792,301,298
	susan (smooth. small)	16,937	2	2	8	8		2	4	30,084,780
	susan (smooth. large)	16,937	2	2	109	109		2	4	349,096,325
	dijkstra_small	11,627	1		29	222		2	5	50,927,674
	dijkstra_large	11,627	1		29	1,177		2	5	244,328,853
	crc (small)	7,438	1	1	1,338	1		2	2	30,235,435
	crc (large)	7,438	1		25,989	1		2	2	587,711,536
	fft (small)	11,036				116		1	5	715,774,279
	fft (large)	11,036				952		1	5	14,400,682,487
	sha (small)	8,710	1	1	306	1		2	3	13,254,951
	sha (large)	8,710	1	1	3,173	1		2	3	137,975,708
	string search_small	7,139				57		1	2	269,813
	string search_large	7,154				1,332		1	2	6,689,371

Tabela 3.3: Contagens de chamadas de sistema para o SPARC V8

`mpeg2encode` do MediaBench abre 15 arquivos, dos quais 12 são arquivos de entrada para seus cálculos. O programa `bitcount` do Mibench chama a rotina `times` 14 vezes. Embora esta rotina seja uma das que possuem problemas de incompatibilidade (comentados na Seção 3.2.7) e retornar tempos sempre nulos, o resultado final dos cálculos não é afetado. A maioria das chamadas a rotinas de S.O. são para leitura e escrita de arquivos de entrada e saída.

A última coluna de ambas as tabelas apresenta o número total de instruções executadas para o respectivo programa. Podemos observar que o compilador utilizado (GCC) produz um código melhor para o SPARC V8 do que para o MIPS I, ambos em termos de tamanho quanto de instruções executadas. A única diferença na contagem de chamadas

apresentadas na Tabela 3.2 e na Tabela 3.3 é na rotina `sbrk`. Esta função é usada para alocação dinâmica de memória, portanto é possível concluir que o GCC gerencia de forma diferente a memória para o MIPS I e para o SPARC V8.

O grande número de instruções executados pelos *benchmarks*, produzindo resultados corretos, indicam a corretude da biblioteca de emulação de S.O. e dos modelos MIPS I e SPARC V8 em ArchC.

A Tabela 3.4 mostra os tamanhos dos arquivos de entrada e saída usados pelos programas. Células marcadas com um hífen representam programas que não possuem entrada de arquivo, precisam somente de parâmetros de linha de comando. Inserir as entradas dentro dos programas fonte como vetores só é possível quando os arquivos de entrada são muito pequenos. Como pode ser visto na tabela, este não é o caso da maioria dos programas que usamos. O caso mais interessante foi do programa `crc`. O seu binário possui somente 9.165 instruções e a entrada na versão *large* tem 26.611.200 bytes (725 vezes maior). Este caso não é isolado, também ocorre em outros programas dos *benchmarks* a entrada ser bem maior que o próprio programa.

A corretude dos programas foi verificada comparando as saídas produzidas pelo simuladores com os arquivos produzidos por uma execução nativa em uma máquina x86. Os resultados dos simuladores foram todos corretos [4].

3.4 Conclusões

Neste capítulo foi destacada a necessidade de suporte à emulação de sistemas operacionais na simulação de sistemas embarcados. Com esta funcionalidade implementada, diversos programas já existentes podem ser executados na arquitetura simulada em ArchC, bastando que eles sejam recompilados. Como agora é possível escrever a saída de um

[4] Existe uma diferença de *endian* entre as arquiteturas simuladas e a arquitetura x86. Esta diferença deve ser levada em consideração na comparação dos resultados.

Benchmark	Programas	Entrada (bytes)	Saída (bytes)
MediaBench	adpcm rawcaudio	295,040	73,760
	adpcm rawdaudio	73,760	295,040
	jpeg cjpeg	101,484	5,645
	jpeg djpeg	5,756	101,484
	mpeg2 mpeg2encode	506,880	76,320
	mpeg2 mpeg2decode	34,906	506,880
	pegwit (encrypt)	91,503	91,537
	pegwit (decrypt)	91,537	91,503
Mibench	basicmath_small	–	426,600
	basicmath_large	–	16,465,695
	bitcount (small)	–	625
	bitcount (large)	–	634
	qsort_small	53,437	53,463
	qsort_large	1,572,431	1,572,490
	susan (smooth. small)	7,292	7,233
	susan (smooth. large)	110,666	110,607
	dijkstra_small	29,144	1,342
	dijkstra_large	29,144	6,931
	crc (small)	1,368,864	42
	crc (large)	26,611,200	42
	fft (small)	–	116,210
	fft (large)	–	970,409
	sha (small)	311,824	45
	sha (large)	3,247,552	45
	string search_small	–	3,197
	string search_large	–	92,672

Tabela 3.4: Tamanhos das entradas e saídas para os programas considerados

programa em arquivo, a verificação de corretude do programa e da própria modelagem da arquitetura ficou facilitada.

Descreve-se uma metodologia genérica que pode ajudar a incorporar a funcionalidade de emulação de sistemas operacionais a qualquer simulador de arquiteturas, servindo tanto para os criados por uma ADL ou manualmente.

A emulação de chamadas ao S.O. pode ser usada para validar os modelos de processadores descritos para ADLs, pois uma quantidade arbitrariamente grande de dados de entrada podem ser usados para testar cada instrução. O método proposto pode ser usado para permitir *benchmarks* maiores no desenvolvimento com ADLs.

A modelagem é feita em duas partes. Uma parte é independente da arquitetura simulada e já é implementada no pacote ArchC. A outra parte, dependente de arquitetura, requer que o projetista implemente pequenas funções de interface específicas para a arquitetura modelada. Esta modelagem faz com que a adaptação da biblioteca de emulação para novas arquiteturas seja muito facilitado.

Rotinas que lidam com *multi-tasking*, como `fork()`, ainda não foram implementadas. Ocorre um erro de compilação caso a aplicação use alguma rotina ainda não implementada.

Capítulo 4

Simulação Compilada e Novas Otimizações

A implementação da técnica de simulação compilada (discutida na Seção 2.3.1) no Projeto ArchC foi importante por ser a primeira implementação de código aberto desta técnica de simulação rápida. Este capítulo discute com detalhes como o conjunto de idéias da simulação compilada, presente na literatura, foi implementado.

Porém, a principal contribuição científica do presente trabalho de doutorado foi a proposta e realização de duas novas técnicas de otimização para que a simulação compilada estática ficasse ainda mais rápida. Estas duas novas técnicas de otimização foram o tema de um artigo publicado no XVI Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD 04) [69] e uma das contribuições de um artigo sobre ArchC publicado em uma edição especial da revista *International Journal of Parallel Programming* [70]. Essas técnicas serão detalhadas neste capítulo.

4.1 Simulação Interpretada em ArchC

A primeira ferramenta desenvolvida no Projeto ArchC foi o simulador interpretado. Esta ferramenta se desenvolveu conjuntamente com a linguagem de descrição de arquiteturas. A técnica de simulação interpretada segue exatamente os mesmos passos do hardware e é

o modo natural de se pensar em um simulador de arquiteturas.

```
1  void sparcv8::behavior() {
2
3    /* Testa se ac_pc é valido ... */
4    if (ac_pc > program_size)
5      cerr << "ArchC Error: Address out of bounds." << endl;
6
7    /* Procura çãinstruo decodificada na cache */
8    if ( dec_cache[ac_pc]->valid )
9      decoded = dec_cache[ac_pc]->entry;
10
11   /* ãNo áest na cache, decodifica */
12   else {
13     fetch = IM->read(ac_pc);
14     decoded = Decode(fetch);
15   }
16
17   /* Executa comportamentos: égenrico, íespecfico do tipo e de instr. */
18   ISA->instruction->behavior();
19   decoded->format->behavior();
20   decoded->instruction->behavior();
21
22 }
```

Figura 4.1: A rotina principal *simplificada* da simulação interpretada

A Figura 4.1 mostra, de forma simplificada, os passos tomados no núcleo da simulação interpretada de ArchC. A descrição de um comportamento de instrução, em ArchC, é feita de forma hierárquica. Primeiro, um comportamento genérico é executado para toda e qualquer instrução. Em um segundo passo, executa-se um comportamento específico para o tipo da instrução. Finalmente, o comportamento específico da instrução é chamado. A partir deste ponto do texto, o conjunto destas três funções de comportamento será chamado simplesmente de função comportamento da instrução. Na Figura 4.1, as linhas 18 a 20 executam esse conjunto de funções.

A desvantagem da simulação interpretada é que ela é lenta para as arquiteturas complexas atuais. Novas idéias foram propostas na literatura para que a simulação atinja um desempenho bastante superior, com a mesma precisão. O conjunto destas idéias leva o

nome de Simulação Compilada. Note, na Figura 4.1, que a implementação da simulação interpretada em ArchC foi feita com uma otimização (linhas 8 e 9): as instruções são decodificadas somente uma vez, pois os campos decodificados são armazenados em uma *cache*. Esta é uma das otimizações propostas na literatura para a simulação compilada.

4.2 Simulação Compilada em ArchC

Logo notou-se a importância de uma nova ferramenta de simulação para o projeto que aproveitasse as novas técnicas disponíveis para aumento de desempenho. Este foi um dos objetivos deste trabalho.

O conjunto de otimizações proposto na técnica de simulação compilada é baseado no fato da aplicação a ser simulada ser conhecida com antecedência. Este fato permite que um simulador de conjunto de instruções seja especializado para uma aplicação e, com isso, possibilita uma vasta gama de otimizações baseadas em um pré-processamento do programa de entrada.

O programa fonte é compilado usando um cross-compilador (**gcc** em nossos experimentos) configurado para gerar um arquivo binário para a máquina alvo. Este arquivo, junto com a descrição ArchC da arquitetura alvo, é a entrada para a nova ferramenta de geração de simulador compilado estático chamada **accsim**. A Figura 4.2 apresenta o fluxo de geração de um simulador compilado usando esta ferramenta. O **accsim** realiza um pré-processamento da aplicação binária de entrada, conforme descrito a seguir, e gera um simulador da arquitetura alvo, na linguagem C++, especializado para execução deste programa. Após a compilação, o novo simulador está pronto para ser executado. A técnica de simulação compilada empregada neste trabalho, incluindo as novas otimizações, foi denominada *Fast Static Compiled Simulation* (FSCS).

A primeira otimização em relação à simulação interpretada consiste em anular o passo

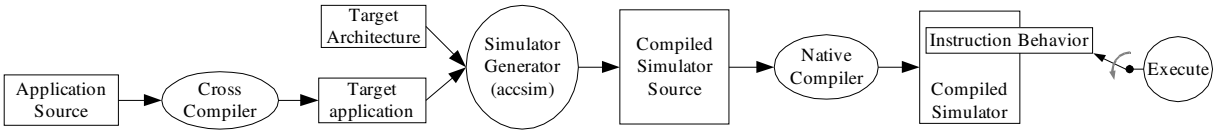


Figura 4.2: Fluxo da Fast Static Compiled Simulation (FSCS)

de decodificação das instruções da aplicação. Já que as instruções são de uma única aplicação, este passo não é mais necessário durante a simulação. Todo o programa a ser simulado é decodificado, instrução por instrução, na fase de pré-processamento, quando é criada uma grande tabela de instruções já com seus campos decodificados. Isso melhora o desempenho da simulação pois cada instrução é decodificada apenas uma vez, independente da quantidade de vezes que ela é executada, o que não ocorre na simulação interpretada convencional.

Outra otimização advém do fato da ordem de execução das instruções também ser fixa, o que permite criar um simulador que escalona o comportamento de cada instrução do programa na mesma ordem original. Cada instrução decodificada do aplicativo corresponde a uma chamada de função comportamento no simulador gerado. Estas funções devem poder ser acessadas aleatoriamente, já que todas as instruções podem potencialmente ser alvo de instruções de saltos. A construção **switch** da linguagem C foi escolhida para indexar estas funções comportamento, ao invés da **goto**, por oferecer uma melhor estruturação e facilidade de leitura do código. Cada uma delas recebe um rótulo **case**. A seleção é feita pelo Contador de Programa (*Program Counter*, PC).

Esta construção **switch** pode ficar bem grande, já que existe um **case** para cada instrução do programa simulado e não se impõe nenhuma restrição para o tamanho deste programa. Isto pode causar um impacto considerável no tempo e no uso da memória na compilação do simulador. A solução encontrada foi quebrar o grande bloco em pequenas construções **switch** que cobrem regiões do programa. Um outro nível de **switch** foi usado

para seleção da região atual.

Uma região é formada por um grupo de 512 instruções e fica contida em sua própria função. A Figura 4.3 mostra uma função de região gerada automaticamente na linguagem C a partir de uma descrição SPARC V8. O laço principal de simulação irá selecionar a região em que se encontra a próxima instrução a ser executada, sendo apresentado na Figura 4.4.

```
1  void Region3() {
2
3    while (1) {
4      switch(ac_pc) {
5
6      ...
7
8      case 0x1954: // be PC-348
9        ac_behavior_instruction(4);
10       ac_behavior_Type_F2B(0, 0, 1, 2, -348);
11       ac_behavior_be(0, 0, 1, 2, -348);
12       ac_instr_counter++;
13       break;
14
15     case 0x1958: // andcc %28, 4, 0
16       ac_behavior_instruction(4);
17       ac_behavior_Type_F3B(2, 0, 17, 28, 1, 4);
18       ac_behavior_andcc_imm(2, 0, 17, 28, 1, 4);
19       ac_instr_counter++;
20       break;
21
22     ...
23
24     default:
25       if ((ac_pc >= 4096) && (ac_pc < 6144)) {
26         AC_ERROR(... non-decoded memory location ...);
27         ac_stop(EXIT_FAILURE);
28       }
29       return;
30     }
31   }
32 }
```

Figura 4.3: Função de região gerada para arquitetura SPARC V8

O simulador gerado com a técnica FSCS é estático, baseado na hipótese que a aplicação

```
1  void Execute(int argc, char *argv[]) {
2
3    model_syscall.set_prog_args(argc, argv);
4
5    while (!ac_stop_flag) {
6      switch(ac_pc >> 11) {
7
8      case 0: Region0(); break;
9
10     case 1: Region1(); break;
11
12     ...
13
14     case 14: Region14(); break;
15
16     default:
17       AC_ERROR(... print error and debug info ...);
18       ac_stop(EXIT_FAILURE);
19       break;
20     }
21   }
22 }
```

Figura 4.4: A rotina principal da simulação compilada

não sofre mudança em seu código binário durante a simulação. Esta hipótese é usada para gerar um simulador rápido que terá a aplicação codificada internamente.

As funções que correspondem aos comportamentos das instruções são chamadas com os campos das instruções decodificados como argumento. Estas funções são marcadas ainda como *inline*, o que permite a um compilador otimizador gerar um código mais rápido para o simulador.

A técnica apresentada é o simulador compilado básico, que ainda será otimizado na próxima seção. Mesmo assim, comparado com simuladores similares presentes na literatura, os resultados obtidos foram superiores. Um grupo de pesquisadores da Universidade da Califórnia, em um trabalho recente [71], afirma ter o simulador redirecionável mais rápido no nível de instrução. A técnica proposta por eles é a *Instruction Set Compiled Simulation* (IS-CS). A Figura 4.5 compara o desempenho de ambas as propostas usando

uma máquina similar (Athlon 1 GHz). A FSCS atinge uma melhoria do desempenho de 90% a 170% apenas utilizando o simulador básico.

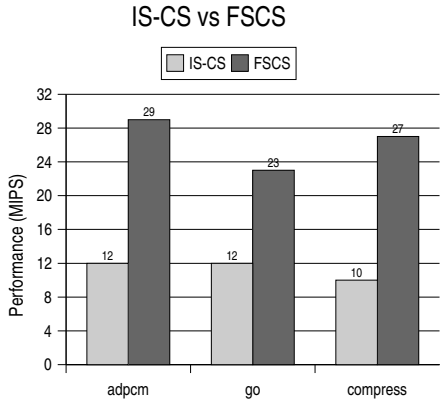


Figura 4.5: Comparação entre IS-CS e FSCS

Outro projeto, da Universidade de Aachen, na Alemanha, denominado *Just in Time Cache Compiled Simulation*(JIT-CCS) [48] atinge 95% do desempenho da simulação compilada. Isso permite uma comparação do JIT-CCS com o FSCC. O site do projeto [72] informa que o desempenho alcançado é de até 30 MIPS (milhões de instruções por segundo) em um Athlon 2.5GHz. Os resultados deste trabalho, apresentados na seção 4.3.3, mostram que a FSCS alcança no mínimo 45 MIPS para o simulador gerado para arquitetura MIPS.

4.3 Otimizações para o simulador básico

É um grande desafio ter um modelo que é eficiente em dois aspectos: qualidade da descrição e desempenho do simulador gerado. Para se conseguir uma descrição de alta qualidade, o modelo deve capturar facilmente a informação da arquitetura de uma maneira natural e concisa para uma ampla gama de arquiteturas. Por outro lado, para gerar um simulador de alto desempenho, reduzindo as operações que o simulador precisa realizar

em tempo de execução, o modelo deve prover o máximo possível de informações estáticas sobre a arquitetura e seu conjunto de instruções.

Para obter um melhor desempenho do simulador, foram incluídas novas construções na linguagem de descrição ArchC que fornecem mais informações sobre o processador. O uso das novas construções é opcional. As informações adicionais, quando presentes, permitem a geração de um simulador mais eficiente. Este trabalho propõe dois níveis de otimização baseados em informações incrementais sobre a arquitetura dadas pelo desenvolvedor.

4.3.1 Otimização 1

Para aumentar o desempenho da simulação compilada, deve-se reduzir a computação realizada em tempo de execução. Uma das formas encontradas foi reduzir a frequência com que o fluxo de execução do programa simulado é avaliado.

Um bloco básico[1] pode ser simulado sem nenhuma verificação por mudanças no Contador de Programa. São normalmente poucas as instruções da arquitetura que influenciam no fluxo do programa. Apenas estas instruções devem ser monitoradas para detecção de saltos. Portanto, elas devem ser identificadas junto com sua descrição em ArchC.

Como a granularidade do simulador compilado é instruções, a propriedade de *delay slot* presente em certas instruções deve ser tratada com cuidado. O fluxo de execução, neste caso, pode mudar somente após a execução da instrução no *delay slot*. Por esta razão, o número de *delay slots* precisa ser fornecido na descrição do conjunto de instruções.

Algumas arquiteturas, como a SPARC V8, causam um problema adicional a ser resolvido: os *delay slots* podem ser anulados. Informações adicionais sobre os casos em que *delay slots* são anulados seriam demais para uma primeira tentativa de otimização. A informação sobre as instruções de controle e sobre o número de *delay slots* é suficiente se o teste por saltos for feito após a instrução de controle e após cada instrução do *delay*

[1] Conjunto de instruções com apenas um ponto de entrada no início do bloco e outro na saída.

slot.

A informação adicional pôde ser integrada facilmente à linguagem ArchC seguindo as mesmas regras de sintaxe. Como as informações são propriedades de cada instrução, foram incluídas palavras-chaves na descrição das instruções de maneira similar a orientação à objetos em C++. As novas construções estão apresentadas em negrito na Figura 4.6 para as instruções *call* e *be* do SPARC V8. As propriedades **call.jump** e **be.branch** foram inseridas na descrição ArchC para indicar que essas são instruções de controle de fluxo. A propriedade **delay** para ambas as instruções indica que cada uma possui um *delay slot*.

```
1  ...
2  call.set_asm("call %disp30");
3  call.set_decoder(op=0x01);
4  call.jump();
5  call.delay(1);
6
7  be.set_asm("be %an, label");
8  be.set_decoder(op=0x00, cond=0x01, op2=0x02);
9  be.branch();
10 be.delay(1);
11 ...
```

Figura 4.6: Novas informações para a Otimização 1 na descrição ArchC

Três propriedades foram adicionadas à descrição de instruções. Duas delas, denominadas **jump** e **branch** informam ao ArchC que se trata de uma instrução de controle. Embora estas duas palavras-chave tenham o mesmo sentido neste escopo, elas foram projetadas para representar instruções de controle de fluxo que são incondicionais (**jump**) ou condicionais (**branch**). A terceira propriedade, **delay**, possui um parâmetro que indica que existe uma ou mais instruções de *delay slot* antes que o fluxo de controle mude.

As descrições ArchC existentes para o SPARC V8 e MIPS I foram incrementadas com essas novas informações e o simulador foi automaticamente criado usando esta técnica de otimização. Esta otimização produz um simulador similar ao já apresentado na Figura 4.3, mas com paradas (`breaks`) apenas após instruções de controle e seus *delay slots*, fazendo

com que o Contador de Programa seja reavaliado mais raramente e, portanto, aumentando a desempenho do simulador.

4.3.2 Otimização 2

Para reduzir ainda mais o número de cálculos efetuados em tempo de execução do simulador, o próximo passo de otimização foi permitir que o simulador calcule o máximo possível do fluxo de execução da aplicação simulada na fase de pré-processamento. Apenas os cálculos dependentes dos dados de entrada são postergados para o tempo de execução. Esta é uma abordagem de Avaliação Parcial [73].

O objetivo do segundo nível de otimização foi ambicioso: delegar a tarefa de controlar e manipular o Contador de Programa para o simulador. Para alcançar este objetivo, a descrição das instruções de controle teve que ser ainda mais refinada do que foi para primeira otimização, agora deixando explícito o modo de manipulação do Contador de Programa. Quanto mais informações sobre a arquitetura estiverem disponíveis, melhor será a avaliação parcial do fluxo de execução. O custo de inserir novas construções e aumentar o tamanho da descrição ArchC de um processador é compensado pelos bons resultados obtidos no desempenho do simulador gerado, mostrados na Seção 4.3.3.

As duas informações dadas anteriormente são usadas: quais instruções manipulam o fluxo do programa e quantos *delay slots* elas têm. Além dessas, o simulador também precisa de mais quatro informações: (1) a condição para o desvio no caso de saltos condicionais; (2) como instruções de salto calculam o seu alvo; (3) a condição para execução de *delay slots* (permitindo assim sua anulação); e (4) o comportamento adicional realizado pelas instruções de controle, se existir (por exemplo, instruções de chamada de rotinas devem salvar o endereço de retorno antes do salto).

Neste ponto, as palavras-chave para as propriedades **jump** e **branch** passam a distinguir as instruções de saltos incondicionais e condicionais, respectivamente. Isto sim-

plifica a descrição da condição de desvio, pois somente saltos condicionais precisam desta informação. Uma nova propriedade **cond** foi criada para este fim. Ela recebe como parâmetro uma expressão matemática que revela como a condição é avaliada para a instrução em questão. Esta expressão pode conter em seus termos qualquer um dos campos da instrução. Podem conter também leituras do banco de registradores ou mesmo da memória. A sintaxe da expressão segue à da linguagem C. Portanto, pode-se usar qualquer um dos operadores desta linguagem.

A informação sobre o cálculo do alvo dos saltos deve ser incluída na descrição como um argumento das propriedades **jump** ou **branch**. Este argumento também é uma expressão, que, do mesmo modo que a expressão de condição, aceita cálculos com os campos da instrução e leituras de registradores ou memória. No caso de instruções de salto relativos ao Contador de Programa, basta inserir a variável de controle `ac_pc` na expressão.

A terceira informação necessária para simular a funcionalidade de algumas arquiteturas modernas é a condição para executar a instrução no *delay slot*. Alguns processadores, por exemplo o SPARC V8, permitem que uma instrução de controle não execute a instrução que normalmente seria executada como *delay slot* do salto. No caso do SPARC V8, existe um campo binário `anull` para anular ou não esta instrução. Para contemplar arquiteturas com esta característica, a condição de execução do *delay slot* deve ser informada, também como uma expressão, em um segundo argumento da propriedade **delay**.

Finalmente, uma nova propriedade, denominada **behavior** foi adicionada para coletar a informação sobre o que a instrução de controle executa além do salto. Todas as arquiteturas atuais possuem ao menos uma instrução de chamada de rotina. Este tipo de instrução deve guardar no banco de registradores (ou na pilha em memória) o endereço do Contador de Programa da próxima instrução, de modo que o fluxo de execução possa voltar a este ponto após a execução da sub-rotina. Esta ação de guardar um endereço de retorno deve ser descrita como um trecho de código na linguagem C e passado como

o argumento da propriedade **behavior**. O comportamento descrito nesta propriedade substitui o comportamento descrito pela construção `ac_behavior`.

```
1  ...
2  call.set_asm("call %disp30");
3  call.set_decoder(op=0x01);
4  call.jump(ac_pc+(disp30<<2));
5  call.delay(1, true);
6  call.behavior(writeReg(15, ac_pc));
7
8  be.set_asm("be %an, label");
9  be.set_decoder(op=0x00, cond=0x01, op2=0x02);
10 be.branch(ac_pc+(disp22<<2));
11 be.cond(PSR_icc_z)
12 be.delay(1, PSR_icc_z || !an);
13 ...
```

Figura 4.7: Novas informações para a Otimização 2 na descrição ArchC

A Figura 4.7 mostra um exemplo da sintaxe adicional para a arquitetura SPARC V8. As novas propriedades e novos argumentos estão em negrito. As instruções *call* e *be* são refinadas em relação à descrição mostrada na Figura 4.6 para a primeira otimização. A expressão para o cálculo do alvo do salto foi inserida como argumento para **call.jump** e **be.branch**, a condição para saltar foi inserida com **be.cond** e a ação de salvar o Contador de Programa em um registrador foi inserida com **call.behavior**.

Vários termos usados nas expressões de condição de salto, cálculo do alvo e condição de execução do *delay slot* são campos da própria instrução. Na fase de pré-processamento, eles são decodificados junto com as instruções, tornando-se constantes. Aplicando a técnica de avaliação parcial nestas constantes, as expressões ficam bastante simplificadas. Durante a execução, o simulador realiza menos cálculos para resolver estas expressões e, portanto, tem um ganho de desempenho considerável. Um exemplo de código do simulador gerado para a instrução **be** pode ser visto na Figura 4.8. Note que as expressões desta instrução, descritas na Figura 4.7, já foram simplificadas. O cálculo do alvo pôde ser feito todo no pré-processamento (lembrando que constantes são resolvidas em tempo

```
1  void Region0() {
2
3      ...
4
5      case 0x150:          //instr: be
6        if (PSR_icc_z) {
7          tmp_pc = 336+(12<<2);
8        }
9        else {
10         tmp_pc = 0x158;
11       }
12       ac_instr_counter++;
13       if (PSR_icc_z || !1) {
14         PRINT_TRACE;
15         ac_behavior_instruction(32);
16         ac_behavior_Type_F2A(32, 0, 17, 4, 34);
17         ac_behavior_sethi(32, 0, 17, 4, 34);
18         ac_instr_counter++;
19       }
20       ac_pc = tmp_pc;
21       break;
22
23     ...
24 }
```

Figura 4.8: Exemplo de código gerado para instrução **be** do SPARC V8

de compilação). Apenas as expressões de condição do salto e condição de execução do *delay slot*, que dependem do registrador da arquitetura `PSR_icc_z`, precisam ser resolvidas durante a simulação.

4.3.3 Experimentos

As descrições ArchC dos modelos SPARC V8 e MIPS I foram atualizadas com as informações necessárias para as otimizações. Apenas as instruções de desvio foram incrementadas com novas informações. Elas são normalmente em pequeno número. O conjunto de instruções do SPARC V8 tem 19 destas instruções e o do MIPS I apenas 12.

Foram utilizados os *benchmarks* dos conjuntos Mediabench [68] e Mibench [58] por representarem bem vários domínios de aplicação. Os resultados foram obtidos a partir de

um Pentium 4 de 2.4GHz rodando Linux. O simulador foi compilado pelo GCC 2.96.

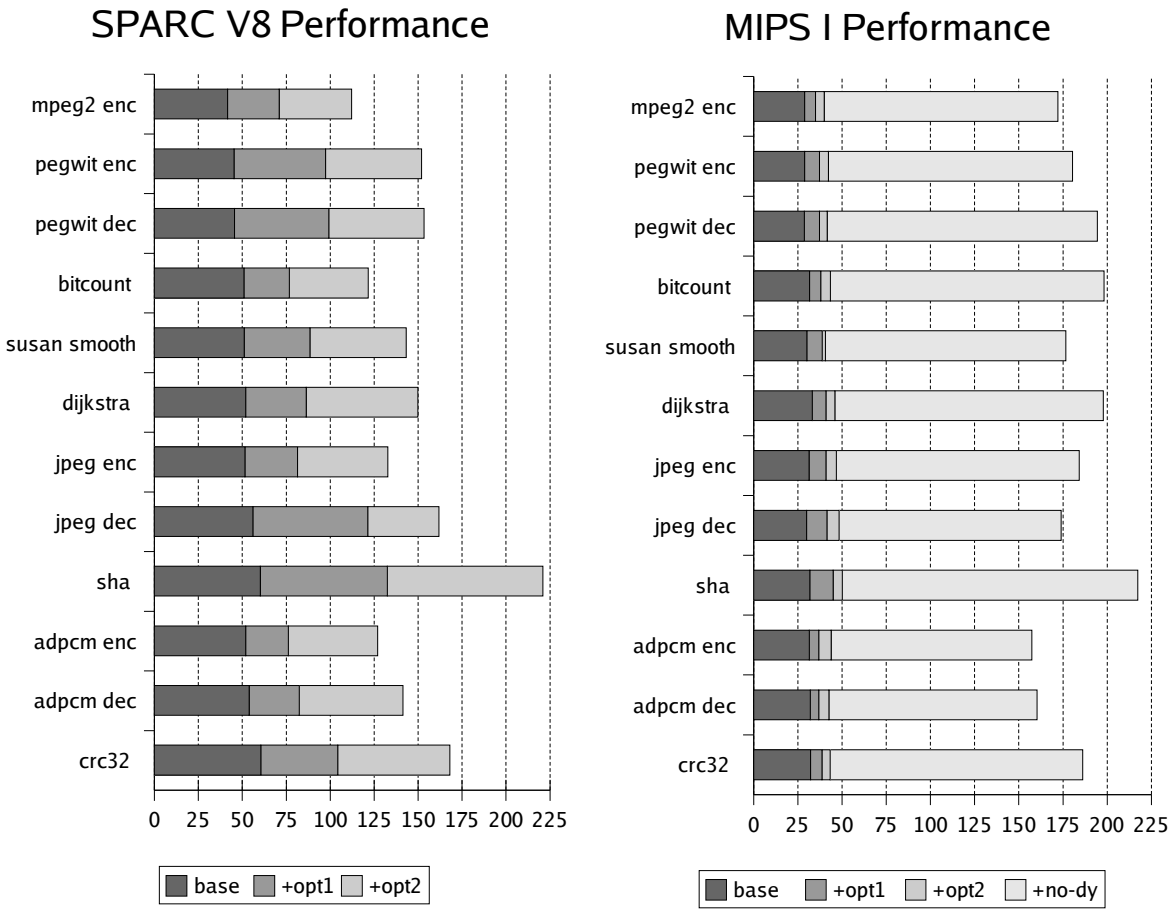
A Figura 4.9(a) mostra os dados de desempenho para arquitetura SPARC V8. Os melhores resultados, da otimização 2, atingem uma faixa de 147% a 244% de melhoria (nos programas *adpcm encoder* e *sha*, respectivamente) em comparação com o simulador compilado base, que já possui um bom desempenho. O programa *sha* apresentou o melhor desempenho na simulação alcançando 210 MIPS, a média ficou em 134 MIPS.

A Figura 4.9(b) apresenta os resultados para arquitetura MIPS I. Os números são menores do que os da arquitetura SPARC V8 porque a implementação do MIPS I em ArchC utiliza um recurso da linguagem chamado *delayed assignment* que permite o atraso de uma atribuição. Quando este recurso está ativado, o simulador analisa a lista de atribuições futuras para efetivar aquelas agendadas para o ciclo atual e isso toma um tempo considerável da simulação.

O modelo MIPS I utiliza o recurso de *delayed assignment* somente para o Contador de Programa, com a finalidade de simular o *delay slot* das instruções de controle. Neste caso, a otimização 2 pôde dar um ganho de desempenho muito grande, pois como o controle do programa passa a ser feito pelo simulador, o recurso *delayed assignment* pode ser desativado. Sem o *overhead* causado por este mecanismo e sem as operações para cálculo do Contador de Programa, o código gerado para o simulador fica mais simples e no mínimo 250% mais rápido. Este ganho de desempenho é representado na Figura 4.9(b) com a legenda "+no-dy".

Para ambas as arquiteturas testadas, a otimização 2 permitiu a geração de um simulador mais simples para o compilador, reduzindo o tempo de compilação em 20% comparado com o simulador base. O tempo de compilação cresce linearmente com o tamanho da aplicação, ficando em 85 segundos para um simulador gerado a partir de uma aplicação de tamanho médio (20.000 instruções).

Um outro experimento foi o de comparar o tempo de execução de um mesmo *bench-*



(a) SPARC V8 (b) MIPS I

Figura 4.9: Desempenho do simulador compilado com otimizações

mark em dois casos: usando os simuladores gerados ou rodando o programa compilado nativamente. Como a comparação é relativa, os números obtidos independem da máquina hospedeira usada. A tabela 4.1 apresenta os resultados medidos com a execução de um simulador SPARC V8 em um Pentium. Também apresentamos os números de desempenho para o mesmo simulador SPARC V8 compilado e executado em uma máquina SPARC nativa na tabela 4.2. A diferença na razão de desempenho para as diferentes arquiteturas nativas pode ser atribuída duas razões. A primeira é que a arquitetura SPARC é um RISC,

enquanto o Pentium é um CISC, portanto, existe uma enorme diferença ente os conjuntos de instruções. A segunda é que a classe de processadores Pentium tem memória *cache* interna maior, portanto, os simuladores fazem menos acessos à lenta memória principal neste caso.

Aplicação	**Intel nativo**	**Simulador SPARC**	**Fator de redução**
adpcm	0.90 s	12.69 s	14.10 x
go	45.87 s	1322.44 s	28.83 x
compress	121.73 s	2233.33 s	18.35 x

Tabela 4.1: Comparação do tempo de execução nativo x simulação na máquina Intel

Aplicação	**SPARC nativo**	**Simulador SPARC**	**Fator de redução**
adpcm	1.96 s	45.33 s	23.13 x
go	112.03 s	3226.21 s	28.80 x
compress	132.90 s	7216.13 s	54.30 x

Tabela 4.2: Comparação do tempo de execução nativo x simulação na máquina SPARC

As duas otimizações descritas neste capítulo são importantes para permitir que arquiteturas mais complexas possam ser simuladas dentro do curto espaço de tempo disponível para o projeto. As arquiteturas estão cada vez mais complexas. Atualmente, os projetistas estão encapsulando dois ou mais núcleos processadores no mesmo *chip*. Os simuladores ArchC devem ser capazes de acompanhar esta nova tendência. O próximo capítulo apresenta os primeiros passos nesta direção.

Capítulo 5

Sistemas com Múltiplos Núcleos Processadores

Desde a introdução do primeiro computador, sempre houve demanda do mercado por mais capacidade de computação. O multiprocessamento simétrico (SMP, do inglês *Symmetric multiprocessing*) é uma tecnologia usada há muito tempo para aumentar o desempenho e a eficiência da computação, dividindo a carga entre vários processadores. O SMP é especialmente eficaz em ambientes *multi-threaded*, onde muitas tarefas (*threads*) precisam ser processadas simultaneamente. A implementação de sistemas SMP sempre foi feita com alguns processadores em *chips* distintos mas conectados em uma mesma placa.

A tecnologia de fabricação atual permite a utilização de mais do que 200 milhões de transistores [54] por *chip*, então, a tendência natural foi de condensar todo um sistema multi-processado em um único *chip*, o chamado processador *multi-core*, e vários projetos se proliferaram desde processadores embarcados até processadores para computadores pessoais e servidores.

Multi-core virou a palavra da moda desde que a Intel, acostumada a melhorar o desempenho dos seus processadores principalmente fazendo projetos com *clock* mais rápido, percebeu que a escalabilidade neste sentido tinha chegado a um limite. As correntes de fuga nos projetos abaixo de 90 nanômetros crescem muito e o processador sofre super-

aquecimento pela potência dissipada. A solução indicada pela Intel foi aumentar a densidade de processamento por área através da incorporação de múltiplos núcleos processadores em um *chip* [74].

Em agosto de 2004, a AMD foi a primeira fabricante de processadores x86 a mostrar um processador de núcleo duplo funcionando. O ganho de desempenho varia conforme a aplicação, mas aquelas otimizadas para processamento simétrico devem contar com um ganho significativo de desempenho. Este processador da AMD é *multi-core* homogêneo, cada núcleo tem registradores distintos e sua própria memória cache.

Nem todos os processadores de múltiplos núcleos estão sendo desenvolvidos com núcleos homogêneos. Um dos exemplos mais conhecidos é o processador Cell [75] que possui um núcleo principal com a arquitetura PowerPC e outros oito núcleos com processadores SIMD[1]. Enquanto núcleos simétricos podem ser adequados para trabalhar com muitos dados, eles podem não ser ideais para trabalhar com mídia, que possui necessidade distinta de tempo-real e constância. Neste caso, talvez o uso de núcleos heterogêneos com algumas especializações seja a melhor abordagem.

O uso de processadores *multi-core* em servidores é certo. A maioria das aplicações escaláveis dos servidores são *multi-threaded* e são, automaticamente, contempladas por processadores *multi-cores*. As necessidades de aumento de capacidade de computação com a complexidade do processamento levam os projetistas para *multi-core*. Servidores com processadores *multi-core* aumentam o desempenho por *chip*, o que pode reduzir os custos de manutenção e melhorar o desempenho por watt.

Uma das estratégias mais agressivas de projeto de processadores com múltiplos núcleos e *multi-threads* é a Throughput Computing [76] da Sun. O primeiro projeto seguindo esta estratégia é a arquitetura Niagara, que estará disponível no começo de 2006[2]. Ela possui

[1]Do inglês: *Single Instruction Multiple Data*
[2]http://www.sun.com/processors/throughput/faqs.html

oito processadores similares a um UltraSPARC II, cada um com quatro *threads*. A ênfase do projeto foi no acesso à memória principal e memória *cache* interna. O desafio para a Sun será a criação de aplicações que consigam gerenciar eficientemente as 32 *threads* disponíveis no *chip*. Se ela conseguir, a expectativa é que outros desenvolvedores a acompanhem em 2007.

No campo de processadores embarcados, que representa a maioria do mercado de microprocessadores [54], a corrida para múltiplos núcleos começou a mais tempo e, atualmente, está ainda mais rápida por exigência da área de redes e comunicações. Embora outras aplicações embarcadas precisarem de processadores de alto desempenho, o aumento da demanda de roteamento de pacotes e infra-estruturas de redes sem fio forçam os projetistas a se esforçarem mais para alcançar o desempenho adequado. Enquanto o advento de processadores de núcleo duplo é novidade no mercado de PCs, a maioria dos processadores embarcados topo-de-linha já possuem, atualmente, pelo menos dois núcleos.

5.1 Simulação de SoCs *multi-core* em ArchC

A tecnologia System-on-Chip (SoC) torna possível o desenvolvimento de sistemas embarcados inteiros em um único chip. Com a disponibilidade de centenas de milhares de transistores em uma pastilha, desenha-se um novo cenário em que SoCs podem ser compostos por diversos processadores especializados interconectados por uma elaborada *Network-on-Chip* (NoC) [77]. Neste cenário, métodos que são capazes de direcionar um processador para uma tarefa específica podem se tornar uma ferramenta importante de desenvolvimento.

Os estudos sobre interconexão de processadores nos SoCs são relativamente recentes, com os primeiros trabalhos datando de 1999 [60]. Ainda existe muita oportunidade para novos estudos. Este é um nicho pouco explorado pelas outras linguagens de descrição de

arquiteturas, apenas um trabalho recente publicado pelos criadores de LISA [62] aborda o assunto, que certamente assumirá grande importância no desenvolvimento de futuros SoCs.

O projeto ArchC foi iniciado em 2001 com o objetivo de facilitar a descrição e a simulação de processadores. Um simulador é gerado automaticamente para cada processador descrito. Até recentemente, a preocupação com a corretude da simulação era o objetivo principal dos esforços. Agora que ambos os simuladores, interpretado e compilado, estão maduros, pode-se pensar em melhorar a integração dos simuladores de processadores com outros simuladores de periféricos.

Iniciou-se o trabalho com sistemas de múltiplos processadores em ArchC em setembro de 2004. Desde as primeiras versões, os simuladores ArchC foram concebidos para trabalharem sozinhos. Este fato faz com que apareçam agora várias limitações do projeto original que devem ser vencidas para que os simuladores passem a fazer parte de um sistema. Elas são as seguintes:

Uso de variáveis no escopo global. A descrição de processadores em ArchC permite que alguns estados internos do processador, ou mesmo armazenamentos temporários para facilitar a descrição de comportamentos de instruções, sejam definidos no escopo global do simulador. Dada uma descrição que possua esta característica, não é possível, pelos padrões atuais, criar um sistema que contenha dois simuladores para essa descrição de maneira confiável. Os estados definidos no escopo global iriam sofrer influência da execução dos dois processadores, causando uma condição de corrida.

Núcleo de simulação egoísta. Durante a simulação de um sistema com vários núcleos, cada núcleo deve possuir uma fatia de tempo para atuar dentro de um ciclo. O núcleo da simulação interpretada em ArchC já segue este conceito, pois executa uma instrução a cada ativação. Já o núcleo da simulação compilada, por questão de desempenho, foi desenvolvido para executar todo o programa quando é ativado. Este fato impede que

a execução de outros núcleos (criados em ArchC ou não) seja feita em paralelo com um simulador compilado.

Comunicação entre núcleos. Enquanto havia somente um núcleo de simulação, a comunicação foi um conceito desnecessário. Mas agora, com a simulação de um sistema, que é, por definição, elementos que se relacionam, torna-se um importante aspecto a ser incorporado.

Temporização do sistema. A temporização mais realista de um sistema é obtida somente com a simulação com precisão de ciclos. Porém, nos estágios iniciais de desenvolvimento, o que se deseja, normalmente, é simular apenas a funcionalidade do sistema. Mesmo sem muitos detalhes na descrição do sistema, deseja-se saber qual das configurações disponíveis será a mais rápida. A contagem de tempo na simulação funcional é dada em número de instruções executadas. A unidade de tempo é equivalente ao tempo de execução de uma instrução, que é aproximado por uma constante. Em ArchC, cada um dos simuladores sabe contar o número de instruções que executa de maneira independente. Quando integrados em um sistema, porém, um núcleo pode ter que esperar pela resposta de outro. Os simuladores funcionais não estão preparados para contabilizar esse tempo ocioso.

Cada um desses quatro aspectos serão analisados e solucionados nas seções seguintes. Como o presente trabalho de tese focou em simulação compilada, este foi o simulador escolhido para fazer a simulação de *multi-cores*. A adaptação do simulador interpretado para os quatro itens acima está além da proposta desta tese.

5.1.1 Uso de variáveis no escopo global

ArchC foi concebida de modo a não impor nenhuma restrição quanto a programação do comportamento das instruções do novo processador. O arquivo de comportamentos é escrito em C/C++ e o desenvolvedor tem total liberdade para declarar, além dos com-

portamentos (obrigatoriamente), outras funções e variáveis auxiliares no escopo global que podem ser acessadas de dentro dos comportamentos e, portanto, facilitar a descrição de processadores complexos. Esta liberdade é uma das principais vantagens de ArchC quando comparada à outras linguagens de descrição de arquitetura, porém, cria o inconveniente de impossibilitar a simulação de um sistema de processadores que definem os mesmos nomes globais. Se houver apenas uma instância de uma variável global e dois ou mais processadores quiserem manipular uma variável global com o mesmo nome, isso causa uma condição de corrida. Além de não ser, na maioria das vezes, o comportamento esperado. Normalmente se quer uma instância da variável para cada instância do processador.

Uma solução para este problema seria não permitir a declaração de variáveis fora das funções de comportamento das instruções. Mas isso exigiria uma reescrita de diversos modelos de processadores já descritos em ArchC e, pior ainda, eliminaria a liberdade que é um dos pontos fortes de ArchC. Esta liberdade de declaração de variável "global" (fora dos comportamentos) foi mantida.

Para se obter diversas instâncias de uma variável, uma para cada processador, existem pelo menos duas alternativas em C++ sem que seja necessário mudar o nome da variável. As duas envolvem a retirada do nome da variável do escopo global.

Uma delas é fazer com que as variáveis globais de uma descrição de processador passem a ser membros de uma classe C++. Esta classe seria uma classe que define todo o processador ou uma classe auxiliar que teria que ser instanciada conjuntamente com a classe que define o processador.

A outra alternativa é criar um escopo de nomes que envolva todo o processador, sendo que cada processador do sistema teria seu escopo. Isso pode ser conseguido em C++ com a construção *namespace*[3].

[3]Todas as declarações que se encontram dentro de um bloco `namespace` são subordinadas ao nome

Ambas as alternativas envolvem mudança no modo de descrever processadores em ArchC. Dentre as duas alternativas, a segunda é mais simples de ser implementada, além de ser a que provoca a menor mudança na descrição do ponto de vista do usuário: envolve somente englobar toda a descrição do processador por uma construção *namespace*. Portanto esta solução que envolve cada processador em um escopo foi preferida.

Um bloco *namespace* é instanciado somente uma vez. Para simulação de vários processadores em um sistema, cada processador deve ser compilado com um espaço de endereçamento diferente, portanto o nome dado ao bloco *namespace* deve variar, mesmo que diversos processadores iguais (com a mesma descrição) sejam usados no sistema. Esta variação é obtida através do pré-processamento oferecido pelos compiladores C++. Os escopos são normalmente denominados $P1$, $P2$, $P3$, e assim sucessivamente, embora o usuário do sistema possa escolher uma denominação para cada processador.

Após a conclusão destas mudanças nos modelos, um teste foi feito para verificar o escopo dos nomes. Um sistema foi feito com dois processadores executando de forma sequencial, pois uma execução paralela tinha o problema de "núcleo de simulação egoísta". Neste teste, foram identificados e solucionados problemas relativos a inserção do bloco `namespace` na descrição dos processadores. Algumas variáveis internas do simulador ArchC tiveram que ser encapsuladas também com um bloco `namespace` para que as rotinas de comportamento de cada processador do sistema pudessem acessar a variável interna respectiva. Um exemplo é o contador de programa, que é uma variável controlada pelo ArchC mas que deve existir para cada processador.

5.1.2 Núcleo de simulação egoísta

A simulação compilada em ArchC, descrita no Capítulo 4, foi desenvolvida com o objetivo de se extrair o máximo de desempenho. Isso envolve o menor número possível de

dado ao bloco. Isso permite que nomes possam se repetir em blocos com *namespaces* diferentes.

verificações ou testes de condições durante a execução do simulador. Ela não foi concebida para dividir o tempo de simulação de um processador com outros processadores ou periféricos. O núcleo da simulação compilada simplesmente não faz nenhum teste entre a execução de uma instrução e da seguinte. Isso faz com que a simulação rode sem interrupções, de forma muito rápida.

Esta abordagem não é mais satisfatória para a simulação de um sistema. O núcleo da simulação compilada não pode ser egoísta. Ele deve permitir que outros núcleos de simulação possam ter a oportunidade de ser ativados, para que a simulação de cada elemento do sistema ocorra em paralelo e que possa existir "comunicação entre núcleos".

Duas abordagens foram testadas para este problema. A primeira foi executar somente uma instrução a cada ativação do núcleo da simulação compilada. A segunda foi manter uma ativação do núcleo executando todo o programa, porém colocar cada ativação em uma *thread* separada da máquina hospedeira.

A primeira abordagem permite que vários núcleos sejam ativados de forma intercalada, criando uma ilusão de execução em paralelo. Após a mudança do núcleo para executar uma instrução por ativação, alguns testes foram feitos para medir o desempenho do sistema por esta abordagem. Em um dos testes, um sistema com dois processadores SPARC V8 e dois MIPS I foi executado com sucesso. Este sistema executou programas independentes em cada um dos processadores, sem nenhuma comunicação entre eles. A Figura 5.1 mostra o código usado para troca de contexto entre os 4 processadores. Não existe restrição quanto aos modelos que podem ser simulados em um sistema. Podem ser vários processadores iguais ou mesmo todos os processadores diferentes. Cada processador possui uma variável (`ac_stop_flag`) que controla se a tarefa do processador respectivo já acabou. Caso ainda não tenha acabado, a rotina `Execute` é chamada para que a próxima instrução seja executada.

A mudança no núcleo da simulação compilada provocou uma queda grande no de-

sempenho do simulador. O desempenho em instruções executadas por segundo por um simulador foi por volta de 40% menor. Este fato já era previsto, já que uma característica da simulação compilada — a execução de todo um programa sem interrupção — não é mais válida.

```
1  int main(int ac, char *av[])
2  {
3    // ... ção inicializaes
4
5    while ((proc1::ac_stop_flag == 0) || (proc2::ac_stop_flag == 0) ||
6           (proc3::ac_stop_flag == 0) || (proc4::ac_stop_flag == 0)) {
7      if (proc1::ac_stop_flag == 0) {
8        proc1::Execute(ac,av);
9      }
10     if (proc2::ac_stop_flag == 0) {
11       proc2::Execute(ac,av);
12     }
13     if (proc3::ac_stop_flag == 0) {
14       proc3::Execute(ac,av);
15     }
16     if (proc4::ac_stop_flag == 0) {
17       proc4::Execute(ac,av);
18     }
19   }
20
21   // ... ção finalizaes
22 }
```

Figura 5.1: Troca de contexto na simulação de sistemas

A segunda abordagem para permitir que núcleos fossem executados em paralelo não envolve mudança no núcleo da simulação compilada, mas sim no núcleo da simulação do sistema. O sistema deve ser instanciado de forma a colocar cada núcleo em uma *thread* diferente da máquina hospedeira.

A Figura 5.2 mostra como fica a rotina principal de um sistema com 4 processadores. Uma *thread* é iniciada para cada processador do sistema, depois aguarda-se o término de cada *thread* para encerrar a simulação.

Esta abordagem tem a vantagem de utilizar a característica *HyperThreading* presente

nos computadores Intel atuais. Usando-se esta abordagem, embora o desempenho de cada processador simulado tenha caído pela metade, as *threads* são executadas em paralelo no Intel e o desempenho do sistema fica apenas 16% mais lento do que a soma dos tempos da simulação separada de cada processador (executado sem troca de contexto). O que pode ser considerado um custo baixo para a execução dos núcleos em paralelo. Esta segunda abordagem foi a preferida.

```
1  int main(int ac, char *av[])
2  {
3    pthread_t id[4];
4    int status;
5
6    /* cria cada uma das threads */
7    pthread_create(&id[0], NULL, proc1::main_thread, NULL);
8    pthread_create(&id[1], NULL, proc2::main_thread, NULL);
9    pthread_create(&id[2], NULL, proc3::main_thread, NULL);
10   pthread_create(&id[3], NULL, proc4::main_thread, NULL);
11
12   /* aguarda encerramento de cada uma das threads */
13   pthread_join(id[0],(void **)&status);
14   pthread_join(id[1],(void **)&status);
15   pthread_join(id[2],(void **)&status);
16   pthread_join(id[3],(void **)&status);
17
18 }
```

Figura 5.2: Uso de *threads* para simulação de sistemas

5.1.3 Comunicação entre núcleos

Não basta que vários núcleos de simulação possam ser simulados em paralelo. Um quesito importante na simulação de um sistema é a comunicação entre os seus componentes. Os simuladores em ArchC não possuem esta capacidade porque não existia com quem se comunicar.

A comunicação entre núcleos de simuladores ArchC é um problema razoavelmente simples de resolver, já que envolve somente aspectos conhecidos e permanece no escopo do

projeto ArchC. Já a comunicação com outros simuladores de processadores e periféricos já é mais complexa e será implementada somente futuramente no ArchC versão 2.0, embora um estudo dos requisitos necessários para isso tenha sido feito e é apresentado na Seção 5.3. Nesta fase, o foco ficou na comunicação entre núcleos da simulação compilada em ArchC.

A simulação compilada está implementada no nível funcional. Para este nível de simulação, o método mais direto de comunicação entre processadores é utilizando um banco de memória compartilhado, por isso ele foi implementado primeiro. Outro método utilizado foi um canal ponto-a-ponto de comunicação entre dois processadores com uma fila de requisições.

Para implementação de ambos os métodos, utilizou-se um mapeamento em memória. O bloco de memória compartilhada foi mapeada no endereço em que a memória local de cada processador acaba (por padrão 5MB). As conexões ponto-a-ponto com fila também foram mapeadas logo após a memória local, uma após a outra e alinhadas pelo tamanho da palavra (4 bytes), de acordo com o código de cada conexão (FILA1, FILA2, etc).

Esse mapeamento em memória exigiu que a classe de armazenamento que representa a memória em ArchC, chamada *ac_storage*, fosse modificada. Um acesso de leitura ou escrita a um endereço além do fim da memória local do processador é repassado para um outro componente: memória compartilhada ou conexão ponto-a-ponto.

Memória Compartilhada

A memória compartilhada tem por padrão 5MB. Portanto, o endereçamento de cada processador ficou extendido a 10MB. Em um sistema real, existiria algum tipo de barramento entre os processadores para acessar a memória compartilhada, mas ele foi abstraído porque a simulação é funcional e também por questão de desempenho.

O primeiro teste da nova infra-estrutura do sistema de processadores com mecanismo de comunicação foi a programação de um modelo cliente/servidor, sendo que o servidor

rodava em um processador e o cliente em outro. Para o servidor, foi escrito um programa simples de cálculo do fatorial de um número dado como entrada. O cliente fazia requisições para o servidor passando alguns números como parâmetro.

Como em todo sistema que executa em paralelo com memória compartilhada, este também apresenta a questão de concorrência de acesso à esta memória global. O problema de concorrência não foi resolvido nesta fase[4], ele será resolvido futuramente. Por hora, evita-se usar endereços compartilhados que possam ser escritos por mais de um processador. Portanto evita-se que qualquer concorrência ocorra. Este objetivo é alcançado com o uso de um protocolo para a comunicação entre os processadores.

A comunicação entre os processadores foi feita por um protocolo assíncrono simples, ilustrado na Figura 5.3, com sinais de controle `request` e `response`, e dados `input` e `output`. O processador cliente coloca o pedido em `input` e ativa o sinal de `request` para pedir um cálculo de fatorial para o processador servidor. Esse, após o cálculo, disponibiliza a resposta em `output` e ativa o sinal de `response`. O cliente coleta a resposta e libera o sinal `request`. Por fim o servidor libera o sinal `response` e fica livre para outro pedido. Como o servidor só escreve em `response` e `output` e o cliente em `request` e `input`, não ocorre problema de concorrência de escrita.

Conexão ponto-a-ponto com fila

O uso de uma conexão ponto-a-ponto com fila para comunicação entre núcleos simuladores, também chamada neste texto de fila de comunicação, é a base para outras conexões mais complexas, como a conexão de um processador com um barramento ou a conexão de um processador com outros processadores formando uma NoC.

Existem alguns trabalhos na literatura que estudam a interconexão das NoCs [77, 60,

[4]Lembrando que como problema do "núcleo de simulação egoísta" foi resolvido com *threads* da máquina hospedeira, ainda pode haver uma concorrência para escrita no objeto que representa a memória compartilhada no sistema simulado no nível interno do simulador. Esta questão foi resolvida usando os mecanismos padrão de exclusão mútua do modelo de *threads* POSIX do Linux.

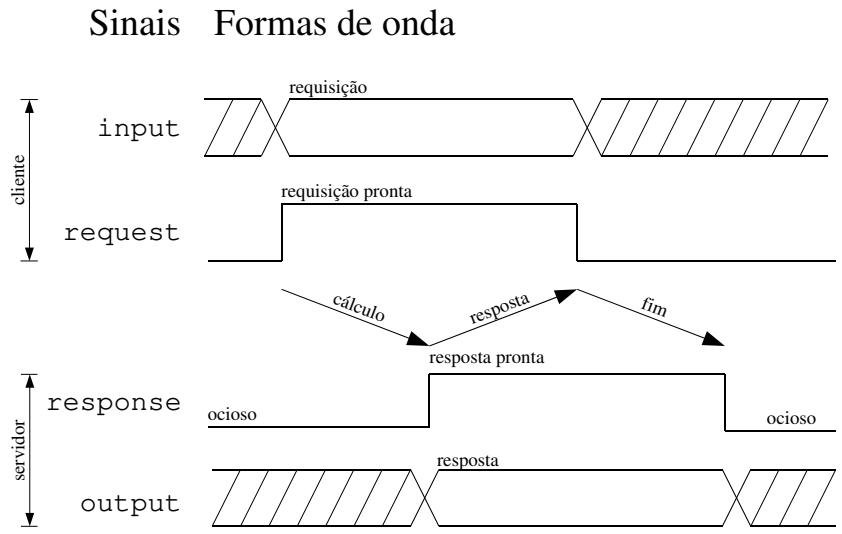


Figura 5.3: Protocolo assíncrono utilizado para comunicação entre processadores

78]. Poucos deles focam na especialização da interconexão da NoC para uma aplicação [78]. O desempenho da aplicação em uma NoC pode ser bastante influenciado pelo modelamento da interconexão dos núcleos. Ao dar suporte ao modelamento genérico da interconexão, o simulador de sistemas ArchC permite o estudo dessa influência.

As conexões ponto-a-ponto com fila fazem uma comunicação em apenas um sentido. Uma ponta da conexão é ligada a um dispositivo X que vai enviar dados, a outra é ligada a um dispositivo Y que vai receber esses dados. No restante do texto, refere-se a este tipo de conexão como sendo no sentido X→Y.

A aplicação cliente/servidor foi modificada para utilizar fila de comunicação. Neste caso, duas conexões são necessárias, chamadas de FILA1 e FILA2. A FILA1 foi conectada no sentido cliente→servidor, para envio da requisição pelo cliente, e a FILA2 no sentido servidor→cliente, para resposta do servidor.

Abstração da comunicação

A escolha do método de comunicação é feita durante a configuração do sistema. Os programas executados nos processadores não devem sofrer alterações só porque o modelo de comunicação do hardware foi alterado.

Uma camada de abstração da comunicação foi implantada. Isso possibilita que o mesmo programa, sem modificação, seja compilado e executado de forma independente da conexão do sistema. A abstração consiste em duas funções, uma chamada `ac_send()`, para enviar dados, e outra `ac_recv()`, para receber dados. Ambas as funções possuem três parâmetros: o código do dispositivo fonte/destino, o buffer de leitura ou escrita e o tamanho da mensagem. Estas funções são fornecidas como uma biblioteca para serem usadas pelo projetista do software.

5.1.4 Temporização no simulador de sistema

O simulador de sistemas descrito realiza corretamente a simulação de sistemas, porém, tem a desvantagem de não contabilizar o tempo decorrido no sistema.

A contagem de tempo na simulação funcional é dada em número de instruções executadas. A unidade de tempo é equivalente ao tempo de execução de uma instrução, que é aproximado por uma constante. Em ArchC, cada um dos simuladores sabe contar o número de instruções que executa de maneira independente.

Pode-se pensar que o tempo decorrido no sistema é igual ao número de instruções executadas pelo processador que demorou mais para finalizar sua tarefa. Isso não é verdade por causa do tempo de comunicação entre os processadores. Quando integrado em um sistema, um núcleo pode ter que esperar pela resposta de outro. Este tempo que o processador fica ocioso também deve ser contabilizado. Portanto, o tempo de execução de um processador passa a ser o número de instruções executadas mais o número de

instruções equivalentes ao tempo que ele ficou ocioso. Assim, o tempo decorrido no sistema será tomado como o tempo de execução do processador que levou mais tempo para parar.

A técnica de Relógios Lógicos, de Lamport [79], foi usada para calcular o tempo ocioso dos processadores. A Figura 5.4 mostra a relação de precedência que deve existir entre eventos. Em eventos ocorridos em um mesmo processo é trivial a relação de precedência. Para processos distintos, o evento de envio de uma mensagem deve preceder o evento de recebimento dessa mensagem.

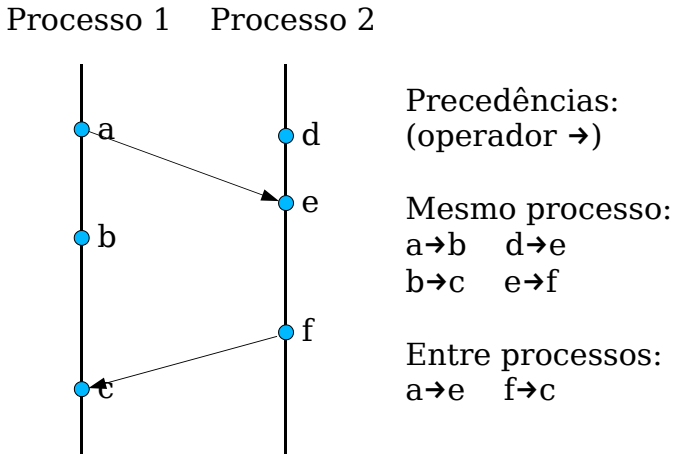


Figura 5.4: Relação de precedência na técnica de Relógios Lógicos

Os Relógios de Lamport colocam etiquetas temporais lógicas nos eventos com números inteiros cuja relação de ordem satisfaz as propriedades da relação de precedência, garantindo que, quando uma mensagem for recebida, a etiqueta temporal do receptor seja superior à do emissor.

O número utilizado para etiqueta no envio de uma mensagem foi o próprio número de instruções executadas no processador até o momento. Portanto, durante a recepção da mensagem, o contador interno de instruções do processador destino é atualizado se, e somente se, o número de etiqueta recebido for maior que o número atual. Esse ajuste

impede que a recepção da mensagem preceda logicamente o seu envio.

Esta técnica permite a contagem de tempo de simulação corretamente, porém, não impõe limite no número de mensagens que podem ser enviadas em um curto espaço de tempo. Tome como exemplo o caso comum de particionamento de uma aplicação entre um produtor e um consumidor. Neste caso, se o produtor enviar rapidamente mensagens com dados e o consumidor demorar para processar, a aplicação simples da técnica de Lamport só será possível se existir uma fila de recepção de tamanho infinito para o consumidor.

Para limitar o tamanho da fila de recepção, faz-se com que o produtor aguarde quando a fila estiver cheia. Isso implica que o processador produtor também pode ficar ocioso em certos momentos e perder a contagem real de tempo se seu relógio não for atualizado.

Note que este problema é similar ao resolvido para o receptor, porém, no sentido inverso: o produtor é que tem que esperar o cliente consumir ao menos um dado da fila. Esse problema é resolvido fazendo com que o evento de recepção de mensagem devolva uma mensagem especial para o produtor. Essa mensagem de resposta é vazia, serve somente para levar a etiqueta com o tempo de recepção. Desta forma, a técnica de atualização do relógio lógico é aplicada não só no sentido produtor $\rightarrow$ consumidor, mas também no sentido consumidor $\rightarrow$ produtor.

O produtor pode enviar dados (mensagens) até o limite do tamanho da fila. Note o exemplo da Figura 5.5, no tempo 35, a terceira mensagem de uma fila de tamanho 2 não pode ser enviada. A partir deste ponto, o produtor só enviará uma nova mensagem se receber uma resposta indicando que o primeiro elemento da fila foi consumido. Matematicamente, seja $Tp(N)$ o instante em que o produtor enviou a mensagem N, $Tc(N)$ o instante em que o consumidor recebeu a mensagem N e enviou a mensagem de resposta, e S o tamanho da fila, então temos a invariante para manter a contagem de tempo global:

$$Tp(N) > Tc(N - S) \tag{5.1}$$

Portanto, a atualização do relógio lógico do produtor ocorre somente após um bloqueio deste pelo consumidor ter demorado muito a responder (e ter deixado a fila encher). Somente neste caso $Tc(N-S)$ será maior que $Tp(N)$ e esse último terá que ser atualizado para manter a invariante. A Figura 5.5 ilustra um exemplo em que o tempo do consumidor teve que ser atualizado, porque este ficou parado do tempo 5 a 15 aguardando uma mensagem, já o produtor teve que parar do tempo 35 a 45 esperando espaço na fila de mensagens.

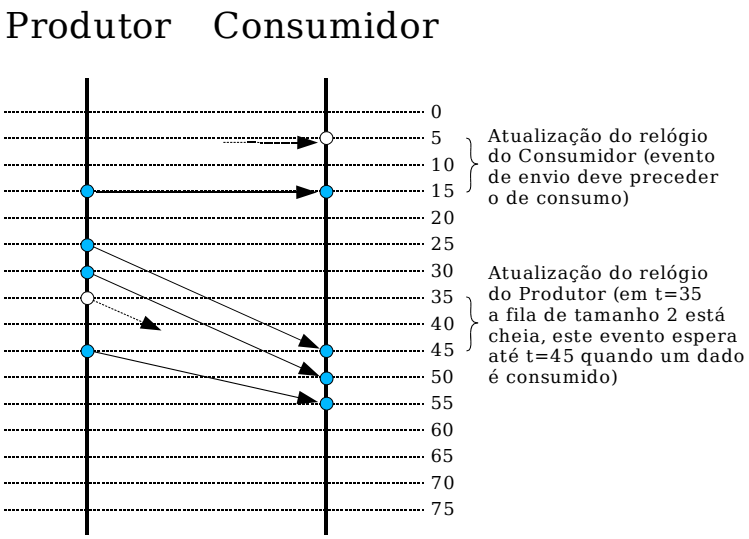


Figura 5.5: Envio de mensagens e bloqueio por fila cheia

A implementação desta técnica permitiu a simulação do tempo global do sistema. Desta forma é possível medir a melhoria de desempenho obtida com o particionamento de aplicações entre vários processadores. A Seção 5.2.1 comenta a metodologia utilizada para o particionamento de programas e as melhorias de desempenho obtidas com ela.

5.2 Software para *multi-cores*

Existem várias maneiras de aproveitar um processador *multi-core*. O uso mais comum é a execução de vários programas em paralelo para executar mais tarefas em menos tempo, todos sobre o controle de um sistema operacional que suporte processadores *multi-core*. Vários sistemas operacionais comerciais já possuem esta capacidade, por exemplo: Microsoft Windows NT, Unix (e similares), Silicon Graphics IRIX, Sun Solaris.

Outra maneira de usar estes processadores é executar múltiplas *threads* em paralelo que pertencem a uma mesma aplicação. Dois exemplos são processamento de transações e aplicações científicas de ponto flutuante paralelizadas manualmente [80]. Neste caso, as *threads* se comunicam usando memória compartilhada e estas aplicações são projetadas para rodar em máquinas paralelas com latências de comunicação na ordem de centenas de ciclos de CPU, portanto, estas *threads* não se comunicam com uma granularidade muito fina. Outros exemplos de aplicações paralelizadas manualmente possuem granularidade fina, de modo que a única forma de explorar devidamente este tipo de paralelismo é usando arquiteturas de múltiplos processadores em um só *chip*, que oferecem baixa latência de comunicação entre os processadores.

O terceiro modo de empregar o multi-processador é acelerar a execução de aplicações sequênciais sem intervenção manual. Isso requer tecnologia de paralelização automática. Recentemente, esta tecnologia se mostrou eficiente para aplicações de propósito científico [81], mas ainda não está madura o suficiente para o uso em aplicações de propósito geral. Da mesma forma que aplicações paralelizadas manualmente, estas aplicações poderiam se beneficiar do aumento de desempenho proporcionado pela baixa latência de comunicação entre os processadores em um único *chip*.

Neste trabalho, a abordagem aplicada foi particionar o programa manualmente, de acordo com a metodologia que será descrita na seção a seguir.

5.2.1 Uma metodologia para particionamento manual

Após a elaboração do simulador de sistemas que contabiliza corretamente o tempo global transcorrido, algumas aplicações são necessárias para testar o simulador. Inicialmente, usou-se aplicações pequenas para validá-lo. Depois, aplicações reais foram executadas para coletar medidas de ganho de desempenho pela execução em paralelo.

Um particionamento ideal de uma aplicação em **N** processadores deveria proporcionar um ganho de **N** vezes no tempo de execução comparado à execução da mesma aplicação em um só processador. Mas isso depende de duas condições: (a) a carga de trabalho em cada partição ser exatamente igual; (b) o tempo de comunicação ser instantâneo. Na prática, existem algumas dificuldades que devem ser levadas em consideração. Um exemplo é o *overhead* causado pela passagem do contexto (variáveis vivas) de um processador para o outro. A comunicação entre os processadores, mesmo que tenha baixa latência como é o caso dos *multi-cores*, ocorre por um protocolo e não é instantânea. Este tempo de comunicação precisa ser levado em consideração para o particionamento da aplicação. Existe um limite a partir do qual não valerá a pena particionar o programa em mais trechos, mesmo que existam processadores sobrando, pois o custo da passagem do contexto provocaria um aumento no tempo total de execução.

Outra dificuldade prática é a não linearidade dos programas. Uma área que estuda muito bem o comportamento dos programas é a de otimização de código para compiladores. Pesquisas nesta área desenvolveram os conceitos de bloco básico, grafo de fluxo de controle (CFG), grafo de fluxo de dados (DFG) e grafo de fluxo de controle e de dados (CDFG) [82]. Através do CFG é possível enxergar o caminho do fluxo de execução, que não é linear nem mesmo para os programas mais simples. Laços são muito comuns e, pela experiência nesta área, são neles onde o programa gasta a maior parte de seu tempo de execução. A não-linearidade dificulta o particionamento em trechos com carga de trabalho

similares.

Os aspectos que devem ser levados em conta no particionamento são:

- Aspecto 1: Divisão justa de trabalho entre as partições
- Aspecto 2: Redução da quantidade de dados transferidos entre processadores

Levando em conta apenas a divisão justa do trabalho, podemos calcular o ganho de desempenho máximo teórico como:

$$G = 1/max(carga) \tag{5.2}$$

onde $max(carga)$ é a percentagem de tempo de execução da partição com mais carga.

Para validação do simulador de sistema, uma aplicação bem pequena foi usada: a ADPCM, que é um codificador de arquivos de som. Este é um programa pequeno, com menos de 100 linhas de código em C, isso permitiu que seu particionamento manual em duas partes e posteriormente em três partes fosse feito sem dificuldade.

Para decidir onde dividir, o primeiro passo foi gerar o CDFG do ADPCM que pode ser visto na Figura 5.6, onde as arestas contínuas representam o fluxo de dados e arestas tracejadas representam dependências de laço. A partir deste ponto, o problema se torna o mesmo que o particionamento de um grafo direcionado. Levando em conta o aspecto 2 dito anteriormente, o grafo deve ser particionado cortando o menor número possível de arestas, pois cada aresta cortada vira um canal de comunicação entre os processadores para passagem da variável/valor correspondente. Note que os ciclos no grafo formados por arestas tracejadas são laços e evita-se dividí-los. O corte mínimo de arestas foi visualmente determinado neste exemplo simples. Ele está marcado com uma linha tracejada grossa na Figura 5.6 como C1 e corta somente duas arestas. O aspecto 1 de divisão justa do trabalho não foi levando em conta para este exemplo de validação do sistema.

Com a ajuda do grafo, o código fonte da aplicação ADPCM foi dividido em dois. Cada aresta cortada foi substituída por um comando de envio na partição que produz o dado e um comando de recepção na partição que recebe o dado.

Compilando e rodando cada parte da aplicação ADPCM em um processador, o simulador de sistemas pôde ser verificado com seu correto funcionamento. O tempo simulado no sistema com dois processadores ficou menor que o tempo com um processador. Porém, como a divisão justa do trabalho não foi levada em conta, a partição do início do programa ficou com uma carga de trabalho significativamente maior que a outra e, portanto, o ganho obtido efetivamente foi de 1.42 (para um máximo teórico 2).

O teste seguinte foi fazer uma interconexão menos trivial entre processadores. Para isso foi feito um outro corte C2, também marcado com uma linha tracejada grossa na Figura 5.6. Chamou-se de partição A a que vai do início do programa até o corte C2, de partição B as operações entre C2 e C1, e de partição C o restante do programa que está após o corte C1. Para esta configuração, a partição B não utiliza o dado *sign* calculado na partição A, portanto, este dado pode ser passado diretamente para partição C, por uma conexão específica entre estas partições. Três canais de comunicação entre processadores são necessários: $A \rightarrow B$, $B \rightarrow C$ e $A \rightarrow C$. A Figura 5.7 mostra como ficou o fonte de cada uma das três partições. Como neste experimento a divisão justa do trabalho ainda não foi levada em conta, esta divisão alcançou o ganho de 1.73 para um máximo teórico 3.

Após a validação do simulador de sistemas, ele pode ser usado como ferramenta de pesquisa. O objetivo é testar diversas configurações de sistema, como processadores ligados a um barramento, ligados por uma *Network on Chip* (NoC) [77] ou ligados com conexões ponto a ponto.

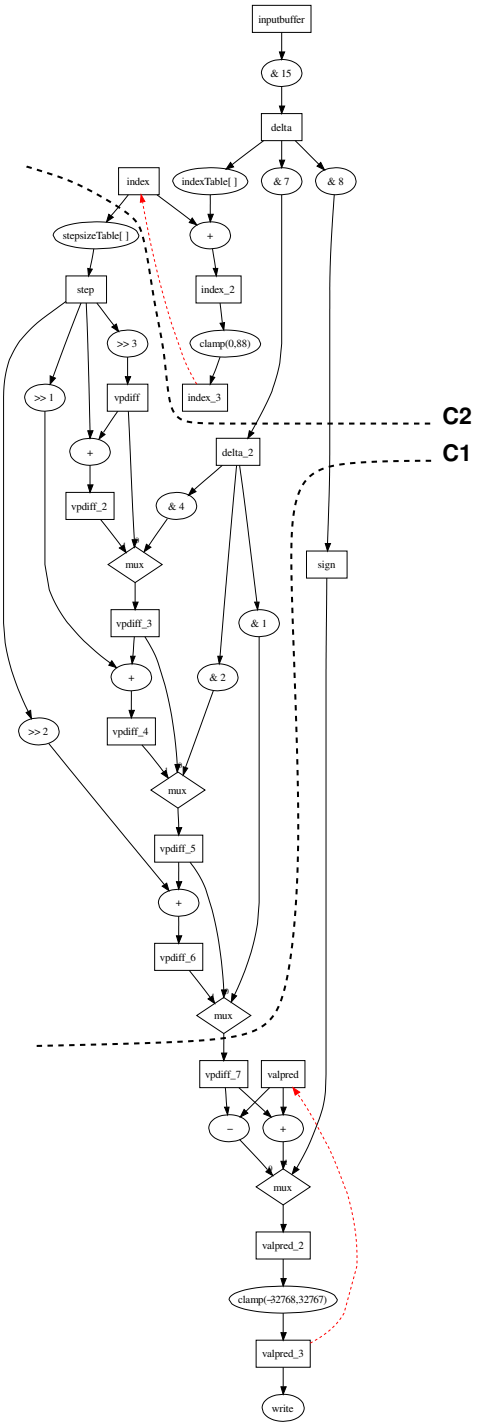


Figura 5.6: Grafo de fluxo de controle e de dados do programa ADPCM

```c
int main(int argc, char** argv) {

  signed char   in;
  int n, sign, delta, step;
  int index=0, bufferstep=0;

  while(1) {

    step = stepsizeTable[index];

    /* Step 1 - get the delta value */
    if ( bufferstep ) {
      delta = in & 0xf;
    }
    else {
      n = read(0, &in, sizeof(char));
      if ( n == 0 ) break;

      delta = (in >> 4) & 0xf;
    }
    bufferstep = !bufferstep;

    /* Step 2 - Find new index value */
    index += indexTable[delta];
    if ( index < 0 ) index = 0;
    if ( index > 88 ) index = 88;

    /* Step 3 - Take sign and magnitude */
    sign = delta & 8;
    delta = delta & 7;

    ac_send(proc3, &sign, 4);
    ac_send(proc2, &step, 4);
    ac_send(proc2, &delta, 4);
  }

  return 0;
}
```

Partição A

```c
int main(int argc, char** argv) {

  int delta;
  int step;
  int vpdiff;

  while(1) {
    ac_recv(proc1, &step, 4);
    ac_recv(proc1, &delta, 4);

    /* Step 4a - Compute difference value */
    vpdiff = step >> 3;
    if ( delta & 4 ) vpdiff += step;
    if ( delta & 2 ) vpdiff += step>>1;
    if ( delta & 1 ) vpdiff += step>>2;

    ac_send(proc3, &vpdiff, 4);
  }

  return 0;
}
```

Partição B

```c
int main(int argc, char** argv) {

  int valpred = 0;
  int sign;
  int vpdiff;

  while(1) {
    ac_recv(proc2, &vpdiff, 4);
    ac_recv(proc1, &sign, 4);

    /* Step 4b - Compute new predicted value */
    if ( sign )
      valpred -= vpdiff;
    else
      valpred += vpdiff;

    /* Step 5 - clamp output value */
    if ( valpred > 32767 )
      valpred = 32767;
    else if ( valpred < -32768 )
      valpred = -32768;

    /* Step 7 - Output value */
    *outp++ = valpred;
  }

  return 0;
}
```

Partição C

Figura 5.7: Três partições do programa ADPCM

5.2.2 Estudo de caso

Um programa bem maior, o MAD, que é um decodificador de audio MPEG Layer 3 (MP3), foi escolhido para um estudo de caso. Esta aplicação tem um tamanho que torna proibitivo aproveitar a mesma abordagem manual de particionamento utilizada com o ADPCM. Criou-se uma nova metodologia para a divisão de programas de tamanho arbitrário.

O primeiro passo desta metodologia consiste em fazer o *profile* de uma execução típica da aplicação. O *profile* deve fornecer dados de desempenho de cada função em relação ao resto do programa e, também, um grafo que indique as chamadas de função realizadas. O grafo deve ser anotado com a informação de desempenho de cada função.

Neste ponto, o projetista, que vai particionar a aplicação, tem a liberdade de escolher em quantas partes ele quer particionar o programa. O número de partes pode ser fixado *a priori* ou o projetista pode observar o comportamento das funções para decidir. Por exemplo, se notar no grafo de chamadas de funções dois conjuntos de funções que desempenham em torno de 50% da aplicação cada um, então ele pode particionar em dois utilizando estes dois grupos. Por outro lado, o projetista/pesquisador pode querer dividir a aplicação de diversas formas, com variações tanto no número de partições quanto nas funções que farão parte de cada partição. O número de combinações possível é exponencial. Ainda existe o caso em que uma só função tenha uma carga de trabalho muito grande e que o projetista decida quebrar esta função e colocar cada parte em uma partição.

Realizou-se o *profile* para o programa MAD. A Figura 5.8 mostra os dados coletados em forma de árvore de chamadas de funções. As funções que gastaram um tempo significativo de execução foram anotadas na figura com a porcentagem do tempo gasto.

O programa foi particionado em duas partes, a que executa o trecho inicial ficou com 63,8% da carga de trabalho e o trecho final com 36,2%. O corte é mostrado na Figura 5.8.

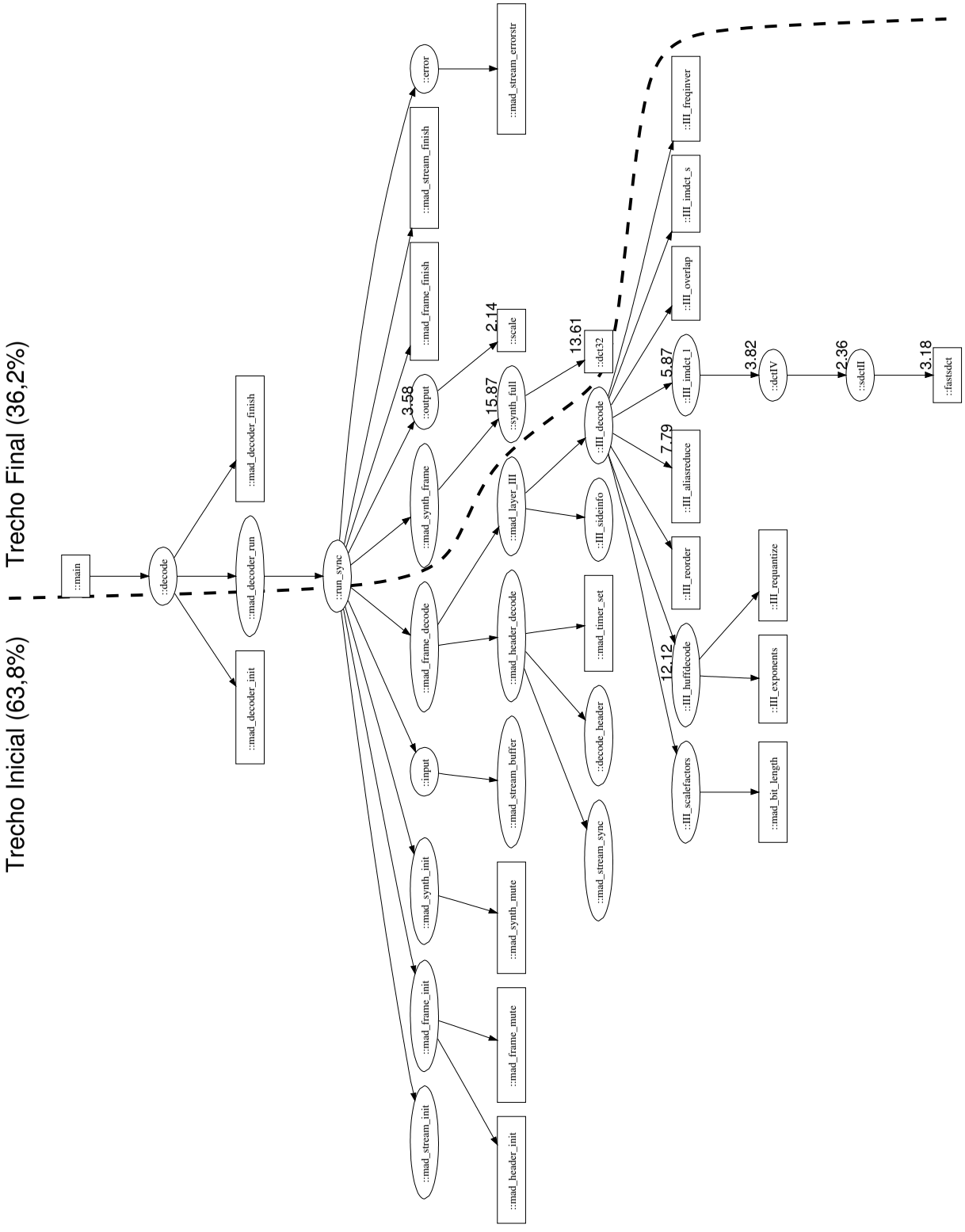


Figura 5.8: Grafo de chamadas do programa MAD

Com esta divisão de trabalho, pode-se calcular um limitante superior para o ganho de desempenho se considerarmos que o sistema não pode ser executado mais rápido do que o maior trecho, de tamanho 63,8% do total. Portanto o ganho de desempenho máximo G dada esta divisão de trabalho é de:

$$G = \frac{1}{63,8\%} = 1,56 \tag{5.3}$$

A MAD particionada em dois foi executada no simulador de sistemas utilizando conexões ponto-a-ponto com filas entre os processadores. O ganho de desempenho em relação à execução em um único processador foi de 1,41, portanto, ocorreu uma perda devido à comunicação entre os processadores em relação ao ganho teórico de 1,56 calculado pela equação 5.3.

A pesquisa de outras maneiras de particionar código para o melhor aproveitamento do paralelismo inerente aos programas será objeto de estudo de um aluno de mestrado do LSC.

5.3 ArchC 2.0: Integração com outros IPs

O simulador de sistemas atualmente é composto por diversos simuladores compilados gerados pelo ArchC, um para cada processador. O gerador de simuladores compilados do ArchC, por questões de desempenho, foi elaborado de forma a não utilizar o SystemC como máquina de simulação. Porém, com a simulação de sistemas, surgiu o desejo de co-simular ArchC com módulos genéricos (IP - Intectual Property), possibilitando o acoplamento de qualquer bloco de hardware descrito em SystemC, como memórias, ASICs, etc, no simulador de sistemas. A ausência de SystemC no simulador compilado impede que IPs em SystemC sejam co-simulados com esse simulador atualmente.

Para que módulos SystemC possam ser conectados ao sistema, alguns experimentos

foram feitos modificando o simulador compilado gerado por ArchC para que ele seja inserido em um módulo SystemC e assim possa aceitar conexões de diversos dispositivos descritos também em SystemC.

Esta conexão deve ser feita seguindo um padrão, uma interface determinada para a comunicação de dados. Esta interface foi desenvolvida baseada na experiência atual do projeto ArchC. Tradicionalmente, estão disponíveis métodos de leitura e escrita para três tipos de dados: byte, palavra (*word*) e meia-palavra (*hword*). Esta funcionalidade foi mantida na nova interface e dois novos métodos foram inseridos: os métodos de leitura e escrita de blocos de bytes de tamanho arbitrário (*block*). A Figura 5.9 ilustra o diagrama UML [83] da hierarquia de classes de interface (classes relativas ao tipo de dado meia-palavra foram removidos da figura para facilitar a compreensão).

Com a interface interna do ArchC definida, módulos genéricos de hardware podem ser conectados. Se eles possuírem interface de dados diferente, basta usar um conversor (*wrapper*) para conexão.

Para criação de um sistema, um processador pode ser instanciado da mesma maneira que um módulo de SystemC, diversas vezes. O projetista tem a liberdade de modificar cada um deles de forma independente, quebrando as conexões originais e associando novos dispositivos de dados, por exemplo. Desta forma, cada processador pode ter sua memória local ou todos podem se conectar a uma memória global. É possível também uma solução híbrida com as duas abordagens citadas, como é o caso de arquiteturas NUMA (*Non-Uniform Memory Access*) [84, 85].

Estas mudanças estão planejadas para a versão 2 do ArchC, que será uma mudança grande e voltada para simulação de vários processadores conectados.

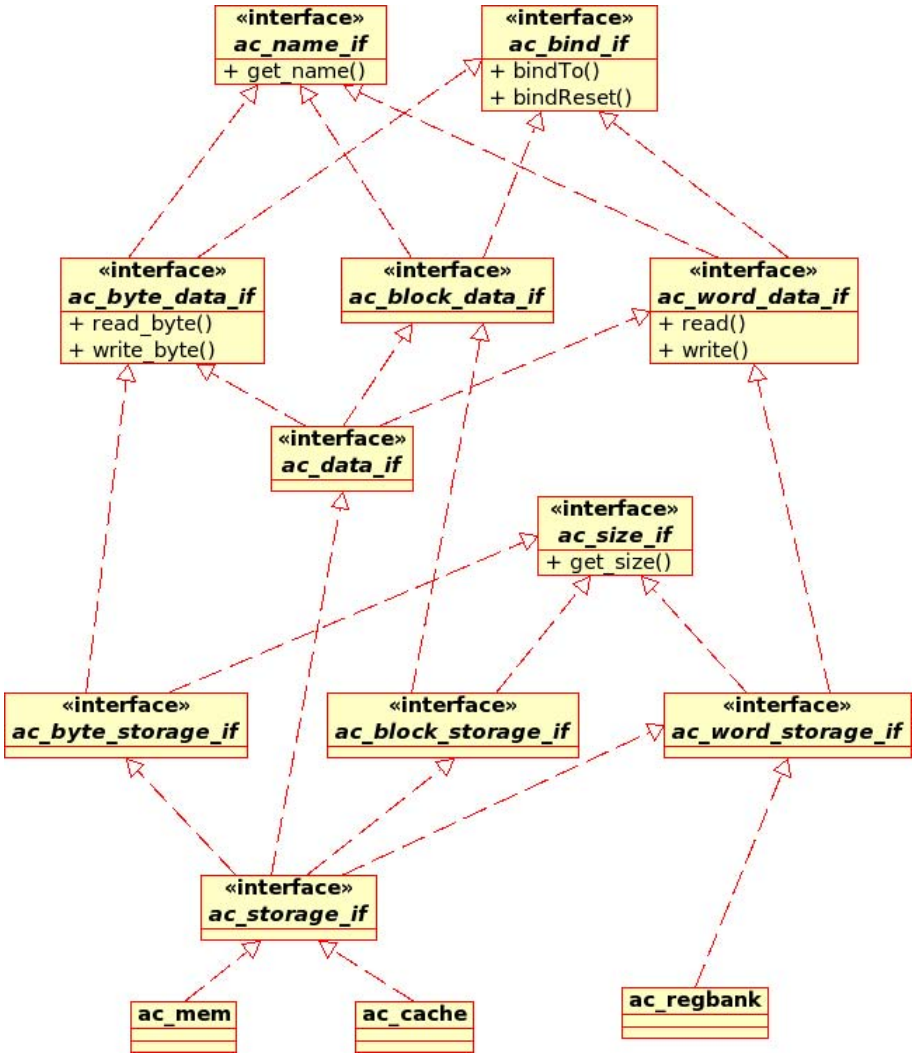


Figura 5.9: Diagrama de classes da nova interface de dados em ArchC

Capítulo 6

Conclusões

Este trabalho de doutorado dotou o Projeto ArchC com uma nova ferramenta de simulação que é muito mais rápida que o simulador interpretado já existente. A diferença no desempenho das duas ferramentas de simulação chega a duas ordens de grandeza. Uma parte deste ganho foi proporcionado pela técnica de simulação compilada já presente anteriormente na literatura. Porém, grande parte do ganho de desempenho foi devido a duas técnicas de otimização inéditas, aplicadas conjuntamente com a simulação compilada.

A implantação dessas novas técnicas, descritas no Capítulo 4, exigiu que a linguagem ArchC fosse expandida para que descrevesse novos aspectos da arquitetura. Com um esforço pequeno do projetista, informações adicionais sobre o modelo ficam disponíveis para o gerador de simuladores compilados (`accsim`). As novas informações tornam possível a geração de um simulador mais eficiente.

Um simulador rápido é importante para reduzir o ciclo de desenvolvimento de uma nova arquitetura. Diversas variáveis devem ser consideradas na criação de um novo projeto de arquitetura computacional, já que cada aplicação tem suas demandas específicas dependendo da área de atuação. Uma simulação rápida proporciona uma melhor exploração do espaço de desenvolvimento (*Design Space Exploration*) sem que o tempo de projeto seja comprometido.

O maior programa já executado em um simulador interpretado é o LAME, presente no *benchmark* Mibench. Esta aplicação é um codificador de áudio para o formato MP3 que executa aproximadamente 8 bilhões de instruções para a entrada de teste. A simulação interpretada alcança o desempenho de 650 mil instruções por segundo em um Pentium 4 rodando a 2.8GHz, portanto, o tempo decorrido para simular esta aplicação foi de quase 3 horas e meia. A maioria das aplicações do *benchmark* SPEC são significativamente maiores, chegando a 231 bilhões de instruções no caso do programa Vortex. O que significaria uma simulação de 98 horas (4 dias) pelo simulador interpretado, para apenas este programa.

Ao utilizar o simulador compilado, o tempo de geração do simulador especializado deve ser levado em consideração no tempo total de simulação. Este tempo de geração é proporcional ao tamanho da aplicação a ser simulada. Para um programa grande como o Vortex, com tamanho de 188.127 instruções, a geração levou 7 minutos na máquina de referência. A execução deste programa levou somente 65 minutos, portanto o tempo total de simulação foi de 72 minutos. Neste caso, o simulador compilado foi 81 vezes mais rápido que a simulação interpretada.

As duas novas otimizações desenvolvidas neste trabalho — simulação atômica de blocos básicos e controle de fluxo pelo simulador — melhoraram bastante o desempenho da técnica de simulação compilada. Comparando a FSCC com outras técnicas presentes na literatura, como a IC-CS do projeto EXPRESSION, que chega a 12 MIPS [71], e a JIT-CCS do projeto LISA, que chega a 30 MIPS [72], a FSCC apresenta um ganho de desempenho substancial, já que passa dos 100 MIPS, como pode ser conferido nas Figuras 4.9(a) – 4.9(b).

Além das otimizações para simuladores compilados, este trabalho traz outras contribuições. A emulação de um sistema operacional foi uma funcionalidade importante adicionada a ambos os simuladores. A metodologia foi criada de forma genérica para

poder ser incorporada facilmente a qualquer técnica de simulação. Essa funcionalidade permitiu um aumento de complexidade dos os programas simulados, que possuíam pouca ou nenhuma entrada e saída de dados, como o `fatorial`. Atualmente, é possível executar programas de tamanho arbitrário, com entradas e saídas complexas, lendo e gravando para vários arquivos ao mesmo tempo e mostrando erros e mensagens no terminal.

Outra área de atuação deste trabalho foi a simulação de sistemas com mais de um processador. Este tipo de sistema é a nova tendência para o futuro, pois os projetos de *chips* com um único processador chegaram a um limite físico de desempenho (esquentam demais). A alternativa encontrada foi criar diversos núcleos processadores, cada um com um poder de processamento menor do que os processadores atuais, porém o desempenho resultante da união desses processadores menores ultrapassa o desempenho dos processadores *single-core*. A simulação desse tipo de sistema permite a exploração de diversos aspectos de comunicação entre eles. Este trabalho iniciou o suporte à simulação de *multi-cores* em ArchC, testando a implementação por *namespace* e elaborando padrões para interface de comunicação de dados. Desde que o trabalho nesta área foi feito, apareceram novos requisitos, como a integração de IPs fechados em SystemC. Estes requisitos serão solucionados com o ArchC 2.0, que usará uma abordagem diferente para simulação de *multi-cores*: cada processador poderá ser instanciado diversas vezes e cada um será um módulo do SystemC. A conexão entre os processadores será delegada ao usuário.

Além das três contribuições citadas (simulação compilada otimizada, emulação de sistemas operacionais e simulação de sistemas multiprocessados), houve algumas outras pequenas contribuições para o Projeto ArchC. As duas mais interessantes foram o suporte a arquivos binários no formato ELF, permitindo que programas reais de grande porte pudessem ser carregados nos simuladores e a correção do decodificador genérico do ArchC para suporte a processadores *little-endian* e a tamanhos de instruções maiores que uma palavra do processador.

6.1 Trabalhos Futuros

Existem diversas possibilidades para extensão deste trabalho. Algumas delas são sugeridas nesta seção.

Completar rotinas de sistema operacional

As rotinas de S.O. implementadas com base na metodologia desenvolvida neste trabalho são suficientes para a execução dos *benchmarks* considerados. Outros programas podem requerer rotinas que não estejam implementadas ainda. Um trabalho simples, talvez no nível de Iniciação Científica, seria completar a implementação destas rotinas seguindo o exemplo das que já estão presentes. O sistema operacional Linux conta com cerca de 250 destas rotinas. Todas elas poderiam ser suportadas pela Biblioteca de Chamadas de Sistemas do ArchC. Por enquanto, um erro de compilação é gerado quando algum programa requisitar o uso de uma rotina não implementada na Biblioteca.

Suporte à simulação compilada com precisão de ciclos

A simulação funcional, que é o nível suportado pelo simulador compilado, é a primeira abordagem que se faz para o desenvolvimento de uma nova arquitetura. A próxima fase do desenvolvimento exige uma maior precisão na simulação do modelo, através da simulação *cycle-accurate*. A maior precisão é obtida com o custo alto na queda de desempenho da simulação. Já existem trabalhos [38] que utilizam a técnica de pré-processamento do programa de entrada, a mesma usada na simulação compilada, aplicada à simulação precisa do *pipeline* de uma arquitetura.

Aplicar a mesma técnica em ArchC traz algumas dificuldades que teriam que ser suplantadas. O gerador de simuladores teria que conseguir informações precisas do *pipeline* de modo a conseguir prever paradas (*stalls*) e *hazards*. A descrição de processadores em LISA é mais extensa do que em ArchC, porém traz a vantagem de conter informações su-

ficientes para prever a maior parte dos estados que o *pipeline* passará durante a execução.

Uma solução seria interpretar o código escrito em C nos comportamentos das instruções e tentar identificar os estágios do *pipeline*. Esta abordagem é bastante complexa. Outro caminho seria incluir novas propriedades para a arquitetura, com o risco de duplicidade de informação com os comportamentos já descritos. Neste caso, informações conflitantes seriam um erro difícil de rastrear no modelo.

A abordagem mais fácil, mas que talvez proporcione apenas um pequeno ganho de desempenho, seria simular o *pipeline* de forma dinâmica, com um controle dos estados do *pipeline* em tempo de execução. O ganho de desempenho viria somente da ordem fixa dos estágios do *pipeline* dentro de um bloco básico.

Opção de simulação compilada dinâmica

Com a crescente exigência do mercado e complexidade dos sistemas, Sistemas Operacionais (SO) estão cada vez mais presentes nos dispositivos. Os simuladores gerados por ArchC suportam emulação de SO. Porém, apenas o suficiente para execução de um só programa por sessão. Uma das funcionalidades do SO é alocação dinâmica de programas na memória. Portanto, para dar suporte a simulação de SO em ArchC, a simulação compilada deve contar com a opção de simulação dinâmica. Neste modo de operação, antes de executar cada instrução deve-se verificar se ela mudou na memória. Isso pode ocorrer em dois casos. O primeiro é quando o SO muda um programa por outro na memória. Outro caso é quando o próprio programa faz uma pequena mudança no próprio código. Caso tenha ocorrido uma mudança no endereço de memória que será executado, não será possível aproveitar os dados da cache de decodificação. A nova instrução terá que ser buscada e decodificada.

O modo dinâmico de simulação traz maior flexibilidade para o simulador, que poderá simular uma gama maior de programas e até um Sistema Operacional completo. Porém,

ele deve ser usado somente quando necessário, já que será invariavelmente mais lento do que a simulação estática por fazer mais verificações em tempo de execução.

6.2 Publicações

A seguir descrevemos as publicações obtidas como resultado do presente trabalho de Doutorado.

- Marcus Bartholomeu, Sandro Rigo, Rodolfo Azevedo e Guido Araújo. *Emulating Operating System Calls in Retargetable ISA Simulators*. Relatório Técnico IC-03-29. Instituto de Computação - UNICAMP. Dezembro de 2003.

- Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo e Guido Araújo. *Optimizations for Compiled Simulation Using Instruction Type Information*. Proceedings of the *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), Foz do Iguaçu-Brazil*. Outubro de 2004.

- Sandro Rigo, Guido Araújo, Marcus Bartholomeu e Rodolfo Azevedo. *ArchC: A SystemC-Based Architecture Description Language*. Proceedings of the *16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04), Foz do Iguaçu-Brazil*. Outubro de 2004. (**Best Paper**)

- Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araújo, Cristiano Araújo e Edna Barros. *The ArchC Architecture Description Language*. *International Journal of Parallel Programming*, Vol. 33, No. 5, Outubro de 2005. DOI: 10.1007/s10766-005-7301-0

É importante salientar que o trabalho intitulado *ArchC: A SystemC-Based Architecture Description Language* recebeu o prêmio Julio Salek Award de melhor artigo da conferência internacional SBAC-PAD em 2004.

Referências Bibliográficas

[1] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[2] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, 1993.

[3] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, N.J., 1992.

[4] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 53–64, Berkeley, CA, 1990. USENIX Association.

[5] Inc. The Santa Cruz Operation, editor. *System V Application Binary Interface – SPARC Processor Supplement*. The Santa Cruz Operation, Inc., 1996.

[6] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrom, and B. Werner. Simics/sun4m: A virtual workstation. In *Proceedings of the Usenix Annual Technical Conference*, pages 119–130, June 1998.

[7] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, February 2002.

[8] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[9] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Alan Herrod. Using the SimOS machine simulator to study complex computer systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[10] Robert C. Bedichek. Talisman: Fast and accurate multicomputer simulation. In *Measurement and Modeling of Computer Systems*, pages 14–24, 1995.

[11] Cathy May. MIMIC: A fast S/370 simulator. In *Proc. of ACM SIGPLAN 1987*, volume 22, pages 1–13, Minnesota, June 1987.

[12] Douglas L. Perry. *VHDL*. McGraw-Hill, 3rd edition, 1998.

[13] Vivek Sagdeo. *The Complete Verilog Book*. Kluwer Academic Publishers, 1st edition edition, June 1998.

[14] Stuart Swan. An Introduction to System Level Modeling in SystemC 2.0. Technical report, Cadence Design Systems, Inc., May 2001.

[15] J. Rowson. Hardware/software co-simulation. In *Proc. of the Design Automation Conference (DAC)*, pages 439–440, June 1994.

[16] Peter Grun, Ashok Halambi, Nikil Dutt, and Alex Nicolau. Rtgen: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS 99 Proceedings*, 1999.

[17] Markus Freericks. The nML Machine Description Formalism. Technical report, Technische Universitä Berlin, Fachbereich Informatiky, July 1993. Updated and Revised Version 1.5(Draft).

[18] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, Neeti Khullar. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *in Proc. Design Automation Conference*, pages 303–306, 1997.

[19] F. Lohr, A. Fauth, and M. Freericks. SIGH/SIM: An environment for retargetable instruction set simulation. Technical report, Fachbereich Informatik, TU Berlin, 1993.

[20] A. Fauth and A. Knoll. Automatic generation of DSP program development tools. In *Proc. ICASSP*, 1993.

[21] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. CHESS: Retargetable Code Generation for Embedded DSP Processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, chapter 5, pages 85–102. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[22] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. ISDL: An instruction set description language for retargetability. In *Design Automation Conference*, pages 299–302, 1997.

[23] Silvia Hanono and Srinivas Devadas. Instruction Selection, Resource Allocation, and Scheduling in the AVIV Retargetable Code Generator. In *in Proc. of 35th Design Automation Conference(DAC)*, June 1998.

[24] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. A methodology for accurate performance evaluation in architecture exploration. In *Design Automation Conference(DAC)*, pages 927–932, 1999.

[25] Norman Ramsey and Jack W. Davidson. Machine descriptions to build tools for embedded systems. *Lecture Notes in Computer Science*, 1474, 1998.

[26] Norman Ramsey and Mary F. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, May 1997.

[27] S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA Language - Version 4.1. Technical report, Computer Science Dpt., University of Dortmund, September 1994.

[28] R. Leupers and P. Marwedel. A BDD-based frontend for retargetable compilers. In *Proc. European Design & Test Conf.*, pages 239–243, Paris (France), 1995.

[29] Rainer Leupers, Johann Elste, and Birger Landwehr. Generation of interpretive and compiled instruction set simulators. In *In Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 1999.

[30] UDL/I Comittee. *UDL/I Language Reference Manual Version 2.1.0a*, 1994.

[31] H. Akaboshi and H. Yasuura. Behavior extraction of MPU from HDL description. In *Proc. of Asia Pacific Conf. on Hardware Description Languages*, pages 67–74, 1994.

[32] H. Akaboshi. *A study on design support for computer architecture design*. PhD thesis, Department of Information Systems, Kyushu University, Jan. 1996.

[33] John C. Gyllenhaal, Wen mei W. Hwu, and B. Ramabriohna Rau. Optimization of machine descriptions for efficient use. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 349–358. IEEE Computer Society, 1996.

[34] P. Paulin, C. Liem, T May, and S. Sutarwala. *Code Generation for Embedded Systems*, chapter Flexware: A Flexible Firmware Developement Envirenment for Embedded Systems, pages 65–84. Kluwer Academic Publishers, 1995.

[35] Ashok Halambi, Peter Grun, Asheesh Khare, Vijay Ganesh, Nikil Dutt, and Alex Nicolau. Expression: A language for architecture exploration through compiler/simulator retargetability. In *DATE99 Proceedings*, 1999.

[36] Asheesh Khare, Nick Savoiu, Ashok Halambi, Peter Grun, Nikil Dutt, and Alex Nicolau. Vsat: A visual specification and analysis tool for system-on-chip exploration. In *EUROMICRO 99 Proceedings*, 1999.

[37] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr. LISA: Machine Description Language for Cycle-Accurate Models of Programmable DSP Architectures. In *36th Design Automation Conference (DAC 99)*, New Orleans, June 1999.

[38] V. Zivojnovic, S. Pees, and H. Meyr. LISA - Machine Description Language and Generic Machine Model for HW/SW Co-Design. In *IEEE Workshop on VLSI Signal Processing*, San Francisco, Oct. 1996.

[39] Chuck Siska. A Processor Description Language supporting Retargetable Multipipeline DSP Program Development Tools. In *Proceedings of International Symposium on System Synthesis (ISSS)*, December 1998.

[40] Sandro Rigo, Rodolfo J. Azevedo, and Guido Araujo. The ArchC architecture description language. Technical Report IC-03-15, Institute of Computing, University of Campinas, June 2003.

[41] Richard P. Paul. *SPARC Architecture, Assembly Language Programing, and C*. Prentice Hall, 2000.

[42] Xilinx. *PowerPC Processor Reference Guide - Embedded Development Kit*, September 2003.

[43] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software - Practice and Experience*, 21(8):877–889, August 1991.

[44] Vojin Zivojnvic, Steven Tjiang, and Heinrich Meyr. Compiled Simulation of Programmable DSP Architectures. In *Proceedings of the 1995 IEEE Workshop on VLSI Signal Processing*, Sakai, Japan, 1995.

[45] Gunnar Braun, Andreas Hoffmann, Achim Nohl, and Heinrich Meyr. Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description. In *Proceedings of the International Symposium on System Synthesis*, October 2001.

[46] Eric C. Schnarr, Mark D. Hill, and James R. Larus. Facile: A language and compiler for high-performance processor simulators. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI01)*, Snowbird, Utah, June 2001.

[47] Jianwen Zhu and Daniel D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of DATE99*, 1999.

[48] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *Proceedings of the Design Automation Conference 2002*, June 2002.

[49] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC2003 Proceedings*, 2003.

[50] Mehrdad Reshadi and Nikil Dutt. Reducing compilation time overhead in compiled simulators. In *Proceedings of International Conference on Computer Design*, October 2003.

[51] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *Procedings of International Conference on VLSI Design*, Jan. 1999.

[52] S. Pees, A. Hoffmann, and H. Meyr. Retargeting of compiled simulators for digital signal processors using a machine description language. In *International Conference on Design Automation and Test in Europe Conference (DATE 2000)*, Paris, March 2000.

[53] Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. A survey on modeling issues using the machine description language lisa. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Salt Lake City, May 2001.

[54] Jörg Henkel. Closing the soc design gap. *Embedded Computing*, 36(9):119–121, September 2003.

[55] Lance Hammond and Kunle Olukotun. Considerations in the design of hydra: A multiprocessor-on-a-chip microarchitecture. Technical Report CSL-TR-98-749, Stanford University Computer Systems Lab, February 1998.

[56] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, December 1996.

[57] Allen Cheng, Gary Tyson, and Trevor Mudge. Fits: Framework-based instruction-set tuning synthesis for embedded application specific processors. In *Proc. of 41st Design Automation Conference*, June 2004.

[58] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, December 2001.

[59] David Seal. *ARM Architecture Reference Manual (2nd Edition)*. Addison-Wesley Professional, 2000.

[60] P. Guerrier and A. Greiner. A scalable architecture for system-on-chip interconnections. In *Sophia Antipolis Forum on MicroElectronics (SAME)*, pages 90–93, France, October 1999.

[61] Srinivasan Murali and Giovanni De Micheli. Sunmap: a tool for automatic topology selection and generation for nocs. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 914–919, New York, NY, USA, 2004. ACM Press.

[62] Andreas Wieferink, Tim Kogel, Rainer Leupers, Gerd Ascheid, Heinrich Meyr, Gunnar Braun, and Achim Nohl. A system level processor/communication co-exploration methodology for multi-processor system-on-chip platforms. In *Proc. of Design, Automation and Test in Europe Volume II*, 2004.

[63] Adan Rose, Stuart Swan, and John-Michel Fernandez. *Transaction Level Modeling in SystemC*. Cadence Design Systems, 2004.

[64] A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proc. European Design and Test Conf.*, pages 503–507, Paris, March 1995.

[65] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *Computer Architecture News*, pages 13–25, June 1997.

[66] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical report, University of Washington, 1993.

[67] Inc. The Santa Cruz Operation, editor. *System V Application Binary Interface – MIPS Processor Supplement*. The Santa Cruz Operation, Inc., 1996.

[68] Chunho Lee, Miodrag Potkonjak, and Willian H. Mangione-Smith. Mediabench: A tool fo evaluating and synthesizing multimedia and communications systems. In *Proc. of 30th Annual International Symposium on Microarchitecture*, December 1997.

[69] Marcus Bartholomeu, Rodolfo Azevedo, Sandro Rigo, and Guido Araujo. Optimizations for Compiled Simulation Using Instruction Type Information. In *Proceedings of the XVI Symposium on Computer Arquitecture and High Performance Computing (SBAC-PAD 2004)*, October 2004.

[70] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araújo, Cristiano Araújo, and Edna Barros. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, 33(5):32, October 2005.

[71] Mehrdad Reshadi, Prabhat Mishra, and Nikil Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *DAC2003 Proceedings*, 2003.

[72] CoWare LISATek. Lisatek product description. http://www.coware.com/products/lisatek_description.php, 2005.

[73] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993. ISBN 0-13-020249-5.

[74] Intel Press Release. Intel has double vision: First multi-core silicon production begins, February 2005.

[75] D. Pham, S.Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, J. Keaty A. Kameyama3, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M.Wang, J.Warnock, S.Weitzel, D.Wendel, T.Yamazaki, and K.Yazawa. The design and implementation of a first-generation CELL processor. In *IEEE International Solid-State Circuits Symposium*, February 2005.

[76] Sun Microsystems. *Introduction to Throughput Computing*, February 2003.

[77] William J. Dally and Brian Towles. Route packets, not wires: on-chip interconnection networks. In *Proceedings of the Design Automation Conference*, pages 684–689, Las Vegas, NV, June 2001.

[78] Yamauchi, Shogo Nakaya, and Nobuki Kajihara. Sop: A reconfigurable massively parallel system and its control-data-flow based compiling method, April 1996. In Kenneth L. Pocek and Jerey Arnold, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines.

[79] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), jul 1978.

[80] Steven Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *In Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[81] Saman Amarasinghe, Jennifer Anderson, Robert French, Mary Hall, Monica Lam, Shih-Wei Liao, Brian Murphy, Chau-Wen Tseng, Chris Wilson, and Robert Wilson. Hot compilers for future hot chips. In *HOT Chips VII*, pages 167–177, August 1995.

[82] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1986.

[83] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, September 1998.

[84] Richard P. LaRowe and Carla S. Ellis. Experimental comparison of memory management policies for NUMA multiprocessors. Technical Report Technical report DUKE–TR–1990–10, 1990.

[85] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park AZ USA, 1989.